EPS Escola Politècnica Superior
UdG

Projecte/Treball Fi de Carrera

Estudi: Enginyeria Informàtica. Pla 1997

Títol: Paral·lelització especulativa de programes sobre un clúster

Document: Memòria

Alumne: Albert Trias Mansilla

Director/Tutor: Teodor Jové Lagunas i Joan Puiggalí Allepuz
Departament: Arquitectura i Tecnologia de Computadors
Àrea: ATC

Convocatòria (mes/any): Maig 2008

Agraïments

En primer lloc vull donar les gràcies als meus tutors, en Joan i en Teo, per l'ajuda i suport que m'han donat al llarg del projecte.

També vull agrair a en Francesc Castro, per la seva inestimable ajuda en tot el referent a l'algorisme *Fast Multipath Radiosity Using Hierarchical Subscenes*.

I finalment als meus familiars i amics pel suport i ànims que m'han donat durant aquest periode.

Índex de continguts

1. Objectius

L'objectiu principal d'aquest projecte és obtenir una millora de rendiment (en temps d'execució) a l'algorisme de gràfics *Fast Multipath Radiosity Using Hierarchical Subscenes* gràcies a l'execució paral·lela especulada que ens permet obtenir el motor d'especulació per a clústers desenvolupat en el grup de recerca BCDS de la Universitat de Girona

Per tal de dur-lo a terme, ens serà necessari estudiar tant l'algorisme com el motor d'especulació i els diferents conceptes que els envolten.

Els passos que seguirem per aconseguir-lo són:

- Preparar l'algorisme per paral·lelitzar-lo amb el motor d'especulació.
    - Implementar l'algorisme amb C, sense utilitzar referències a memòria visibles des de diferents funcions. A més a més, cal evitar la recursivitat, i cal fer el codi el més seqüencial possible, per obtenir una major facilitat a la seva divisió en blocs.
    - Dividir l'algorisme en blocs.
- Adaptar el motor d'especulació a les noves necessitats.
    - Permetre que la mida dels missatges enviats sigui dinàmic.
    - Incloure el graf de blocs de l'algorisme.
    - Implementar el model Mestre-Mestre/Esclau-Esclau amb PVM.
- Augmentar el grau de paral·lelisme de la implementació.
- Obtenir resultats i estudiar quin seria el temps d'execució si disposéssim de més nodes, com també quina serà la tendència en els pròxims d'anys si apliquem la llei de Moore i la de Glider.

2. Introducció

Aquest projecte està emmarcat en l'àmbit de l'arquitectura de computadors i el paral·lelisme.

Concretament consistirà en paral·lelitzar l'algorisme de gràfics *Fast Multipath Radiosity Using Hierarchical Subscenes* [2,4] sobre un clúster Linux aplicant tècniques d'especulació, de dades i de control, mitjançant el motor d'especulació desenvolupat en el grup de recerca BCDS, de la UdG, per Joan Puiggalí, Teodor Jové i altres [10,11,12]. Cal dir que aquest motor encara està en desenvolupament i l'haurem de modificar per tal d'obtenir millores i poder aplicar les tècniques d'especulació sobre l'algorisme esmentat.

En aquest apartat donarem una visió ràpida sobre el paral·lelisme, les arquitectures avançades de computadors, les tècniques d'especulació i el motor citat anteriorment, com també de l'algorisme de gràfics.

2.1. Introducció al paral·lelisme

Quan parlem de paral·lelisme, ens referim al fet de poder dividir una tasca en parts més petites, amb la intenció que diferents unitats de procés puguin resoldre una part del problema total, treballant alhora i sense afectar el resultat final, amb la finalitat d'obtenir un temps de resposta menor.

El paral·lelisme d'una aplicació es pot explotar de diverses maneres: mitjançant les propietats del maquinari (hardware), un compilador o la programació paral·lela. En aquest projecte ens centrarem més en l'aprofitament del paral·lelisme mitjançant software, però aplicant l'ús de les tècniques hardware; per aquest motiu explicarem les hardware.

En un model ideal, la millora del rendiment és lineal en funció del nombre de processadors. D'altra banda en la majoria de problemes reals, ens trobem que totes les parts de l'aplicació no són paral·lelitzables a causa de les dependències de dades i de control. Aquest fragment del programa rep el nom de *part seqüencial* i influeix directament amb el rendiment que es pot obtenir al explotar el paral·lelisme, limitant el nombre d'unitats de procés que podran treballar alhora per tal de resoldre el mateix problema.

La llei d'Amdhal mesura com varia un sistema al introduir-hi millores en funció de la freqüència

d'ús de l'element modificat. També és aplicable al paral·lelisme, on ens explica que si *F* és la fracció de càlcul que s'ha de computar de manera seqüencial i 1-F n'és la paral·lelitzable, llavors el màxim *speedup* (S) que podem aconseguir utilitzant N processadors és el següent:

$$S = \frac{1}{\left(F + \frac{(1 - F)}{N}\right)}$$

Equació 1: Speedup de la llei d'Amdhal

Si observem l'equació i en calculem el límit quan N tendeix a infinit, podem comprovar que mai ens desprendrem de la part seqüencial i que el rendiment obtingut dependrà de la naturalesa del programa. Així mateix, veiem que per tal d'explotar el paral·lelisme d'una aplicació ens interessa que la part seqüencial d'aquest sigui el menor possible.

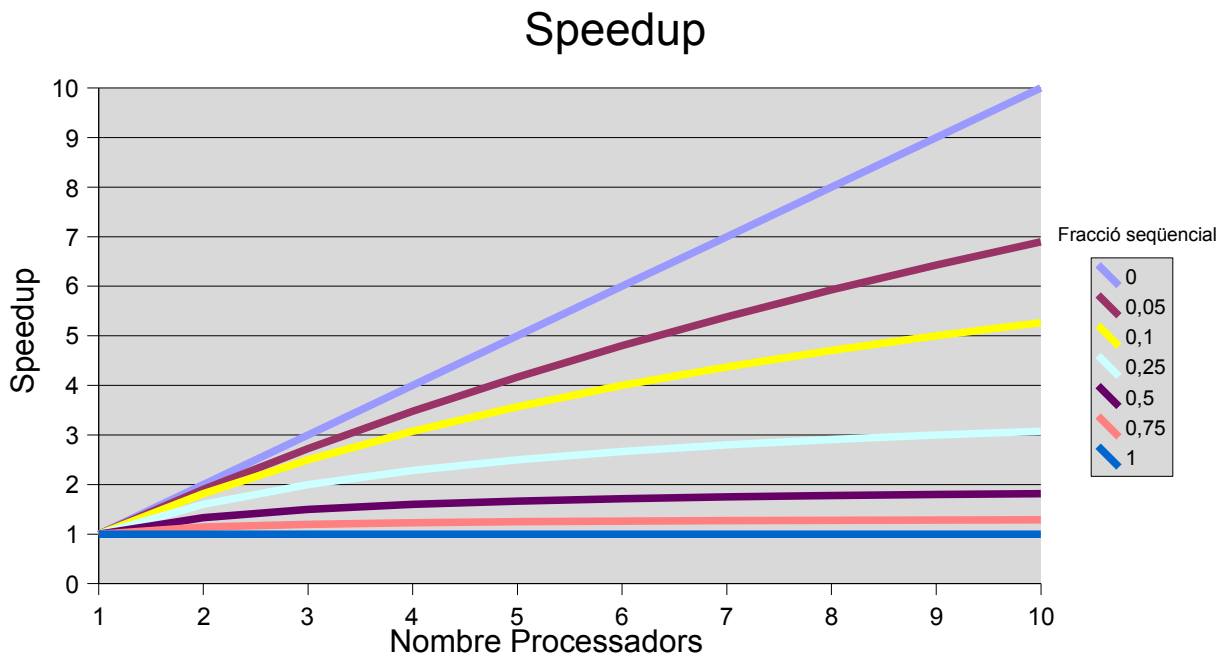


Figura 1: Speedup en funció del nombre de processadors i de la proporció seqüencial

Anteriorment hem esmentat que la causa principal de la fracció seqüencial d'una aplicació ve donada per les dependències. Podem dir que principalment hi ha dos tipus de dependències, les de control i les dades.

Les dependències de control són aquelles que s'originen amb les instruccions de salt, atès que desconeixem quina serà la següent instrucció a executar-se, i ens dificulten continuar l'execució del programa fins que el salt s'hagi resolt.

Les dependències de dades les trobem en les instruccions que necessiten un valor produït en instruccions prèvies. En els processadors actuals ens cal executar la instrucció productora abans de la consumidora, cosa que limita el grau de paral·lelisme per la necessitat de serialitzar alguns fragments del programa.

2.2. Introducció a les tècniques arquitectòniques

L'augment de la velocitat en les màquines actuals es basa en la millora del temps de processador per executar un conjunt d'instruccions. El temps de processador depèn de tres paràmetres.

$$T=CPI*N*T_{cicle}$$

Equació 2: Temps de processador

Observant l'equació, es veu que es pot reduir el temps disminuint el temps de cicle (T_{cicle}), el nombre d'instruccions(N) o els cicles per instrucció (CPI). La reducció del temps de cicle va lligada a les millores tecnològiques com poden ser la capacitat d'integració o l'increment de la freqüència del rellotge, aquest aspecte però té uns límits. La reducció del nombre d'instruccions depèn dels compiladors però és difícil obtenir millores significatives. Finalment ens queda la reducció del nombre de cicles per instrucció, que depèn totalment de l'arquitectura de les màquines; a continuació veurem com el miren de reduir els diferents tipus de processadors.

2.2.1. Processadors segmentats

Els processadors segmentats, es caracteritzen per dividir l'execució de cada instrucció en diversos segments que s'executaran en diferents unitats funcionals especialitzades. Aquesta divisió permet que quan el primer segment d'una instrucció s'ha acabat d'executar, es pugui començar a executar el primer segment de la següent, d'aquesta manera s'aconsegueix que diverses instruccions s'executin de manera solapada en el temps, així s'augmenta el nombre d'instruccions executades per unitat de temps, malgrat no es redueixi el temps d'execució d'una instrucció.

A l'hora d'aplicar aquesta tècnica es fa necessari sincronitzar la durada de les etapes, de manera que una instrucció no passi a la següent etapa fins que totes les instruccions finalitzin l'etapa actual, d'aquesta manera es garanteix que la unitat funcional necessària estarà lliure. Per aquest motiu, s'assigna a la durada d'un cicle de rellotge el temps que triga el segment més lent en executar-se. En el disseny d'una segmentació es procura que el temps en el que les unitats funcionals resten ocioses sigui el menor possible, i s'aconsegueix equilibrant al màxim la duració de les etapes.

En el supòsit que una instrucció tingui les etapes següents:

Cerca d'instrucció (F)	Descodificació/Lectura (L)	Execució (X)	Escriptura (W)

Figura 2: Possibles etapes d'una instrucció segmentada

	1r cicle	2n cicle	3r cicle	4t cicle	5é cicle	6é cicle	7é cicle
Instrucció 1	F	L	X	W			
Instrucció 2		F	L	X	W		
Instrucció 3			F	L	X	W	
Instrucció 4				F	L	X	W

Taula 1: Exemple d'execució d'un processador segmentat

La mitjana del nombre de cicles per instrucció depèn del nombre d'instruccions, i es pot observar que la primera instrucció trigarà tots els cicles, i les que el segueixen en trigaran només un cicle des de la finalització de la primera, fet que permet deduir la següent equació:

$$N_{\text{ciclesmitjà}} = \frac{\left(N_{\text{cicles/instrucció}} + \text{Instruccions} - 1\right)}{\text{Instruccions}}$$

Equació 3: Nombre de cicles mitjà per l'arquitectura segmentada

Es pot veure que quan el nombre d'instruccions tendeix a infinit, de mitjana finalitza una instrucció per cicle.

Cal constatar que és molt difícil equilibrar de manera perfecte les varies etapes, i en conseqüència el rendiment que s'obté és menor al teòric, a més l'exemple mostrat és ideal i no hi ha cap tipus de dependència.

2.2.2. Processadors superescalars

La característica principal dels processadors superescalars és que permeten iniciar l'execució de diverses instruccions en el mateix cicle. Per aquest motiu necessiten la replicació de les unitats funcionals i la introducció de noves vies de comunicació.

Com a nombre de vies, s'entén el nombre d'instruccions que poden iniciar l'execució alhora en un moment donat. L'objectiu d'aquesta arquitectura és dividir encara més el nombre mitjà de cicles per instrucció:

$$CPI_{Superescalar} = \frac{CPI_{Segmentat}}{N^{o}vies}$$

Equació 4: Nombre mitjà de cicles per instrucció dels superescalars

En contrapartida aquesta arquitectura encara és més sensible a les dependències que la segmentada, ja que en executar més instruccions alhora, és molt més probable que s'intentin executar instruccions que tinguin alguna dependència de dades, o que es trobi amb un salt (dependència de control) del que se'n desconeix el resultat i en conseqüència quines seran les següents instruccions a executar.

Per tal d'alleujar aquestes limitacions s'apliquen tècniques de planificació dinàmica, en què el processador decideix en temps d'execució quines instruccions s'inicien en cada instant de temps, amb l'objectiu d'ocupar el màxim nombre d'unitats funcionals en tot moment.

La planificació dinàmica permet que les instruccions s'iniciïn en ordre i finalitzin en desordre (o que tant l'inicialització com la finalització siguin en desordre) i funciona de la següent manera:

En primer lloc, quan s'inicia l'execució d'una instrucció es descodifica, moment en el qual es coneixen les unitats funcionals i les dades que requereix, a continuació les instruccions s'aniran executant a mesura que disposin de les dades necessàries i les unitats funcionals requerides, sempre i quan la instrucció no afecti al resultat de les instruccions que la precedeixen i que encara no s'han executat.

Tot i que existeixen diverses tècniques de planificació dinàmica, en aquest projecte ens basarem en l'algorisme Tomasulo.

2.2.2.1. Tomasulo

En aquest algorisme, la detecció de conflictes i el control de l'execució es troben distribuïts en les diferents unitats funcionals. Cada una d'elles té un nombre determinat d'estacions de reserva, les quals són com petites memòries que permeten reservar la unitat funcional per a diferents instruccions que l'han d'utilitzar. La primera instrucció de les estacions de reserva que disposi dels operands serà la primera en executar-se en la unitat funcional.

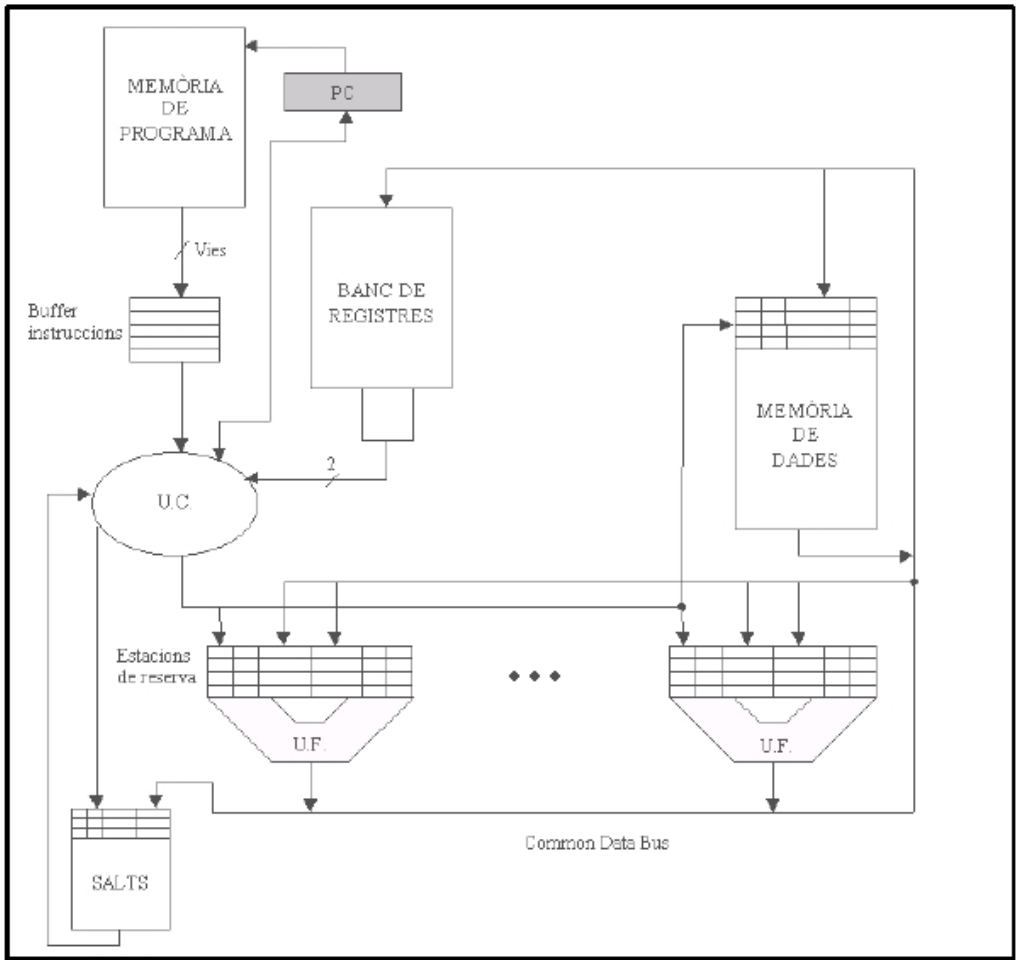


Figura 3: Esquema del Tomasulo

A més, disposa d'un bus comú de dades (CDB) que permet agafar els resultats directament del bus sense esperar que es trobin en el banc de registres. El CDB pot tenir tan sols un port d'escriptura, cas en què serà un bus simple i només hi podrà escriure una unitat funcional a cada cicle; o pot tenir un port d'escriptura per cada unitat funcional, d'aquesta manera hi podran escriure totes les unitats funcionals a cada cicle i serà un bus múltiple.

Per tal d'explicar el funcionament de l'algorisme, és necessari explicar les dades que cal emmagatzemar per cada estació de reserva i per cada registre.

Per cada estació de reserva s'emmagatzema:

- Un **identificador** únic per tal de distingir-lo de qualsevol altra estació de reserva.
- La **operació** que realitzarà la unitat funcional.
- El **valor** dels **operands font** (V_j i V_k)

- Les **estacions de reserva** que produiran els operands font (Q_j i Q_k). En el cas que alguna d'aquestes dades sigui 0, indicarà que el valor de l'operand font ja el tenim emmagatzemat.

- Un indicador que ens mostri si l'estació de reserva està **ocupada o lliure**.

Per cada registre tindrem un camp Q que indica l'estació de reserva que produirà el seu nou valor. En el cas que el seu valor sigui 0 voldrà dir que cap unitat funcional està generant un resultat per a aquest registre.[1]

El Tomasulo necessita tres etapes per a cada instrucció que són l'emissió, la execució i l'escriptura .

A la primera, es busca una unitat funcional que pugui executar la instrucció i que tingui alguna estació de reserva lliure, si no se'n disposa de cap aquesta etapa s'aturarà, per totes les instruccions, fins que se n'alliberi una, d'aquesta manera l'emissió serà en ordre. A continuació, es marcarà com a ocupada l'estació de reserva i actualitzarem les dades de l'estació de reserva, és a dir, si es disposa del valor dels operands en el banc de registres s'emmagatzemaran a V_j i V_k i es posarà un 0 a Q_j i Q_k, en cas contrari Q_j i Q_k indicaran l'estació de reserva que produirà el resultat, cosa que s'obté consultant el camp Q_i del registre font, també caldrà especificar el tipus d'operació. Finalment es marcarà el camp Q_i del registre destí amb l'identificador de l'estació de reserva associada a la instrucció.

A l'etapa d'execució, es busca a cada unitat funcional una estació de reserva que tingui tots els operands i s'inicia l'execució de la operació. En el Tomasulo, les dependències RAW (Read After Write) es resolen en aquesta etapa, ja que fins que no es disposi de tots els operands la instrucció no hi arribarà. Cal dir que aquest algorisme no impedeix que se segueixin emeten i executant més instruccions encara que s'hagi detectat una dependència RAW.

A la darrera, les unitats funcionals que generen un resultat l'escriuen al CDB, des d'on s'actualitzaran els registres que tinguin com a Q_i l'identificador de l'estació de reserva finalitzada i els valors V_j i/o V_k de les estacions de reserva que estiguin esperant aquest resultat.

Aquest algorisme permet que les instruccions es puguin emetre sense esperar que el valor dels operands es trobi al banc de registres, a més a més elimina els conflictes WAW (Write After Write) i WAR (Write After Read) mitjançant les estacions de reserva i el camp Q dels registres.

[1] Ens referirem a aquest camp Q com a Q_i on la i denotarà el número del registre.

2.3. Introducció a l'Especulació

Com s'ha vist anteriorment (punt 2.1), tot programa té una part que s'ha d'executar de manera seqüencial a causa de les dependències entre instruccions que n'impossibiliten el tractament en paral·lel.

És possible aplicar algunes tècniques per mirar d'alleujar les dependències i obtenir un major grau de paral·lelisme, per introduir-les però, ens és necessari parlar dels diferents tipus de dependències. Principalment n'hi ha tres, les de nom, les de control i les de dades.

Les dependències de nom són les que s'originen quan els valors que generen un conjunt d'instruccions s'han d'escriure a la mateixa posició de memòria o al mateix registre. Cal dir que aquesta dependència es pot resoldre variant el destí de les instruccions, i que els microprocessadors actuals són capaços de renombrar els registres dinàmicament per tal d'evitar-les.

Les dependències de control són les que causen les instruccions de salt, i que impossibiliten conèixer quina serà la següent instrucció a executar-se fins que no es resolgui el salt. Per reduir-ne l'afecte es pot aplicar l'especulació de control, que es basa en predir les instruccions de salt.

Les de dades són fruit de les instruccions que consumeixen un valor produït per una instrucció prèvia. Fins que les instruccions productores no s'han executat, les instruccions consumidores no ho podran fer. Aquestes dependències es poden reduir d'una manera semblant a les de salts, és a dir predint el resultat de les instruccions productores.

Tot seguit parlarem de les tècniques especulatives, ja que són les que poden resoldre les dependències de salt i de dades, abans però, necessitem introduir els diferents estats en què es pot trobar l'arquitectura quan l'acabament de les instruccions no és en ordre. Quan s'està fent una execució especulativa el processador es pot trobar en tres estats diferents en funció del contingut del seu banc de registres. Quan els registres contenen els valors assignats per una seqüència d'instruccions consecutives finalitzades, rep el nom d'**estat en ordre**, es tracta d'un estat segur i cap instrucció que el causa es pot desfer. Si en canvi és una seqüència d'instruccions, on n'hi ha alguna de no finalitzada rep el nom d'**estat arquitectònic**. L'**estat look-ahead** conté els valors de l'estat arquitectònic que no estan continguts a l'estat en ordre.

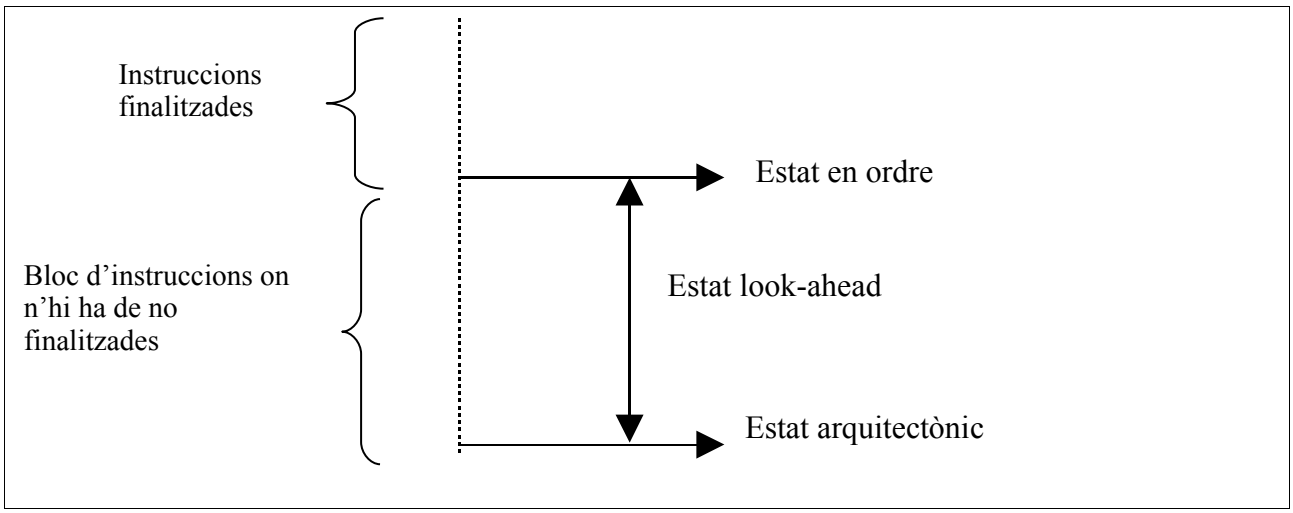


Figura 4: Diferents estats dels processadors

Les tècniques especulatives ofereixen una solució per alleujar el problema originat per les dependències i permeten augmentar el grau de paral·lelisme d'una aplicació. Principalment aquestes tècniques tenen dos factors que afecten al seu rendiment, el primer és l'*overhead* causat pel hardware i el software addicional que requereixen aquestes tècniques; el segon és el cost que suposa fallar l'especulació i que depèn principalment de la taxa d'encert i del cost de recuperar l'estat en ordre.

Les tècniques d'especulació es poden classificar segons el següent esquema:

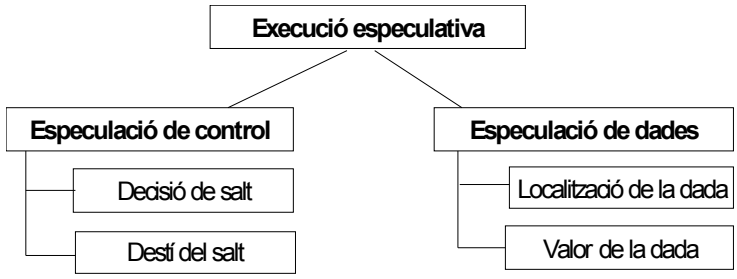


Figura 5: Esquema del tipus de dependències

Tot seguit en veurem algunes.

2.3.1. Especulació en el flux de control

Les instruccions de salt poden variar la seqüència d'execució d'un programa i poden fer que la següent instrucció de la memòria de programa no sigui la pròxima en executar-se. Cosa que dificulta l'emissió de noves instruccions, ja que fins que no finalitzi el salt, es desconeix quines seran les següents.

Quan una instrucció de salt finalitza, dóna dos resultats, l'adreça destí del salt i l'avaluació de la condició. Aquests valors, però, no es produeixen alhora.

Les etapes d'una instrucció de salt condicional poden ser les següents:

Cerca	Descodificació	Càlcul de l'adreça	Avaluació de la condició

Figura 6: Possibles etapes d'una instrucció de salt condicional

Si l'arquitectura segmentada i superescalar esperen a la finalització de les instruccions de salt per a seguir-ne emetent, el seu rendiment pot veure's força minvat. Per tal de reduir l'inconvenient afegit per aquestes instruccions s'han pensat diverses tècniques (com ara la predicció fixa, la predicció dinàmica i la predicció especulativa) per seguir emetent instruccions quan apareix un salt.

Aquestes tècniques es basen en fer una **predicció de la instrucció de salt**, és a dir, suposar si es produirà o no el salt i continuar la seqüència d'execució en funció de la predicció. En el cas que es falli la predicció caldrà anular les instruccions incorrectes i executar les adequades, motiu pel qual es necessita un sistema que anul·li les instruccions que no s'haurien d'haver executat; mentre que si s'encerta la predicció s'hauran avançat unes quantes instruccions. Aquest tipus d'execució rep el nom d'execució especulativa. Quan s'han d'anul·lar algunes instruccions, el que es fa és tornar a l'**estat en ordre,** just després del salt**.**

Per tal de millorar el rendiment de la predicció dels salts es fa necessari avançar el màxim possible el càlcul de l'adreça de destí, per tal de començar a executar tan aviat com sigui possible les instruccions posteriors al salt; com també l'avaluació de la condició, ja que redueix el cost de la recuperació en el cas d'un error de predicció.

La predicció fixa consisteix en assumir sempre el mateix resultat per a una instrucció de salt. Aquest mètode es fonamenta en què les instruccions de salt obtenen el mateix resultat gairebé sempre que s'executen. Es pot implementar de tres maneres, la més senzilla és que en tots els salts sempre es faci la mateixa predicció; la següent que depengui del tipus de salt, és a dir, pels bucles sempre es farà una mateixa predicció mentre que per les condicions se'n pot fer una altra; la darrera, és que el compilador inclogui en el codi generat uns bits que indiquin la decisió a prendre per a cada salt.

La predicció dinàmica pretén adaptar-se en temps d'execució, per aconseguir-ho guarda un històric dels darrers salts.

Tot seguit parlarem del BTB que és un mètode que usa la predicció dinàmica, i del Reorder Buffer que ens permet tornar a l'estat en ordre en cas que hi hagi un error de predicció.

2.3.1.1. Anticipació del càlcul de l'adreça de destí: El BTB

El BTB[2] és un mètode de predicció de salts que usa la predicció dinàmica i pretén guardar l'adreça de destí dels salts que s'executen, atès que pel principi de localitat és molt probable que es faci el salt a la mateixa adreça, un exemple d'aquesta situació són els salts dels bucles. Si s'emmagatzema l'adreça de destí, en cas que es faci una predicció de saltar, al conèixer-la, la latència per calcular-la disminueix.

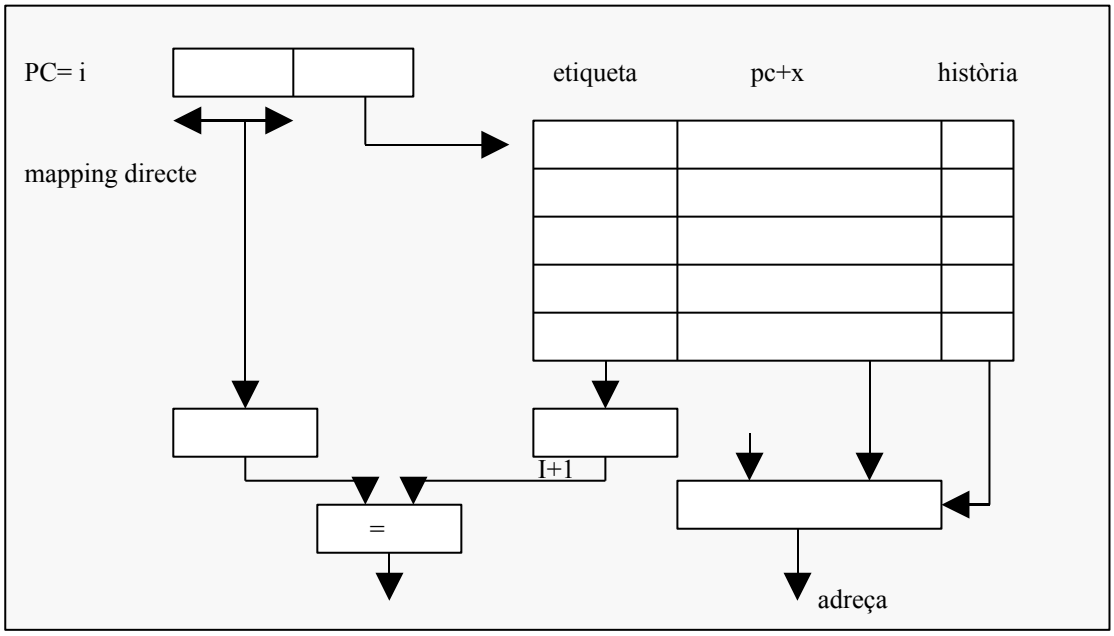


Figura 7: Estructura del BTB

El BTB consta d'una estructura que conté les adreces de destí associades al salt que s'han anat calculant i a més a més l'històric d'avaluació. Com que seria molt costós disposar d'una entrada per a cada posició de la memòria de programa, l'estructura consta a més a més d'una etiqueta que conté els bits més significatius del PC, de manera que s'hi accedeix amb els menys significatius, i l'etiqueta serveix per comprovar que el salt emmagatzemat és el mateix que necessitem.

Aquest mètode consta de dues fases: la predicció del salt i l'actualització de dades.

La predicció del salt serveix per a predir l'adreça de la següent instrucció a codificar. Quan la unitat de control descodifica una instrucció envia automàticament el PC a la memòria i al BTB, on es mira si hi ha una entrada amb els bits menys significatius, si és així es comprova amb l'etiqueta emmagatzemada que coincideixin els bits més significatius per tal de garantir que es tracti de la mateixa instrucció. En el cas es tingui l'entrada per aquest salt, mitjançant l'històric es decidirà

[2] Branch Target Buffer

quina és la següent instrucció a emetre. Si no es conté el mateix salt, es continuarà amb l'execució seqüencial, ja que d'altra manera s'hauria d'esperar al càlcul de l'adreça.

Un cop s'ha avaluat la condició, es procedeix a actualitzar les dades de l'estructura del BTB.

Pel que fa a la penalització del mètode, es suposa que l'accés al BTB no en causa, mentre que la predicció errònia té un cost d'un cicle per a cada instrucció que s'hagi d'anul·lar; a més a més, introduir una instrucció al BTB té el cost d'un cicle.

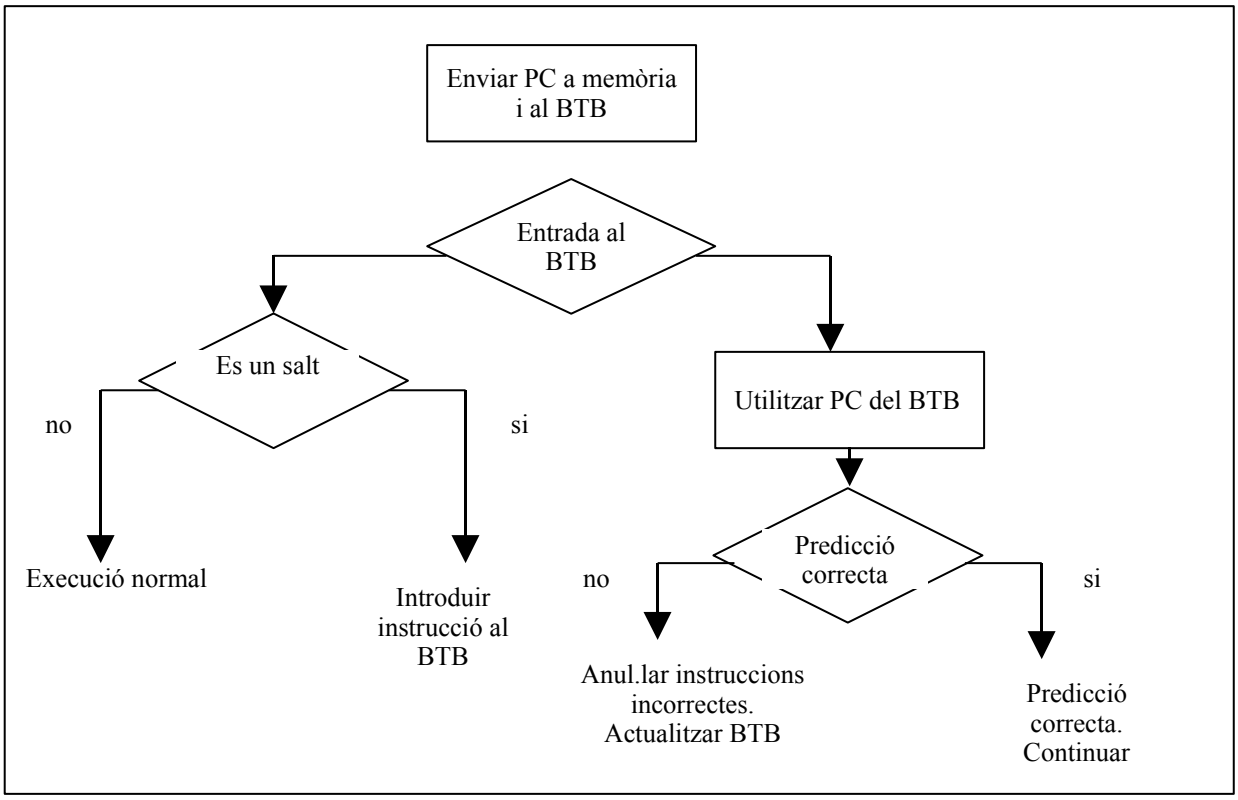


Figura 8: Algorisme del BTB

2.3.1.2. El Reorder Buffer

El Reorder Buffer [18,19] és una tècnica que permet restaurar l'estat en ordre de la màquina en qualsevol moment.

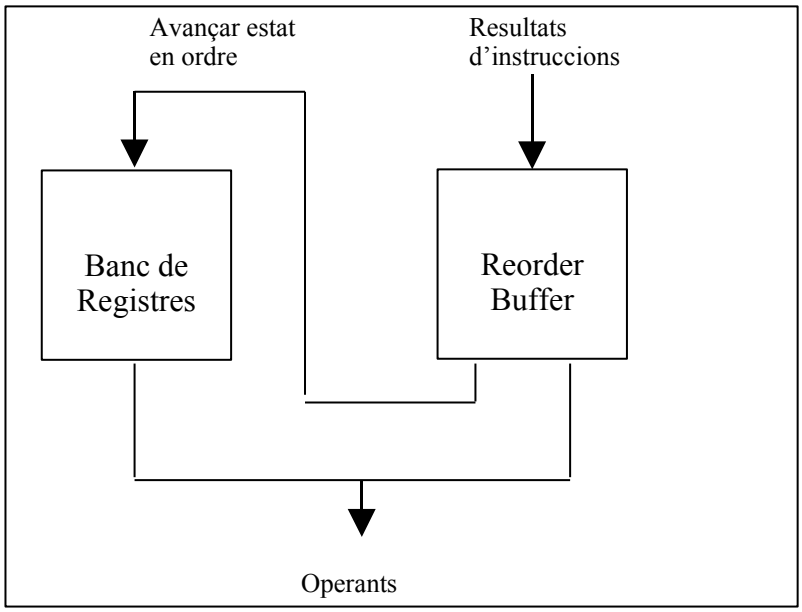


Figura 9: Esquema del Reorder Buffer

El Reorder Buffer és una estructura que conté un camp on s'hi escriurà el resultat de cada instrucció i actua com una cua, on s'hi van encuant les instruccions a mesura que es descodifiquen. Quan una instrucció finalitza, s'actualitza el camp del Reorder Buffer que conté el seu valor. A cada cicle es mira si a l'inici del Reorder Buffer hi ha una entrada amb el resultat, en aquest cas s'actualitza el valor del registre i es treu el primer element de la cua. D'aquesta manera, el banc de registres conté l'estat en ordre i el Reorder Buffer conté l'estat look-ahead[3].

En el cas que no quedi espai disponible en el Reorder Buffer, l'emissió d'instruccions s'atura fins que n'hi hagi.

Quan una instrucció necessita un operand, el seu valor es trobarà o bé en alguna entrada del Reorder buffer o bé en el banc de registres, per aquest motiu la lògica de control del Reorder Buffer ha de ser capaç de trobar el valor més nou del registre desitjat.

En el cas que es produís un error de predicció en un salt, per restablir l'estat en ordre després del salt, caldria esperar a que acabin d'executar-se les instruccions anteriors al salt. Un cop haguessin finalitzat es descartaria el contingut del Reorder Buffer i ja s'hauria recuperat l'estat en ordre. Cal dir que no és gaire normal que un cop finalitzat un salt quedin instruccions anteriors per finalitzar, ja que el salt sol dependre del resultat de les anteriors, motiu pel qual, normalment només caldrà buidar el Reorder Buffer per restablir l'estat en ordre.

El defecte del Reorder Buffer és la seva costosa implementació, ja que requereix memòries associatives prioritzades per buscar el valor més nou d'un registre.

[3] L'estat arquitectònic s'obté de la combinació del banc de registres i del Reorder Buffer.

2.3.2. Especulació de dades

La dependència més freqüent i que causa majoritàriament la serialització de les instruccions és la dependència de dades. Així doncs, l'especulació de dades esdevé un mecanisme prometedor per tal de millorar el grau de paral·lelisme.

Per tal de poder implementar l'especulació de dades, es necessita un mecanisme que permeti al processador predir els valors i la localització de les dades, i que en cas de fallada amb la predicció pugui desfer l'execució de les instruccions errònies.

Aquest tipus de tècniques especulatives sorgeixen de l'estudi de les seqüències de dades. Les seqüències de dades es poden classificar en : *constants*, *strides* i no *strides.*

Les **seqüències constants** són les més simples i provenen, com el seu nom indica, d'instruccions que produeixen repetidament el mateix resultat.

Les **seqüències *stride*** es donen quan els valors successius difereixen entre ells per una constant *delta*. A l'exemple anterior, la *delta* seria 1 que és el valor més comú en la majoria de programes, malgrat que la delta pot prendre qualsevol valor. Una seqüència constant també es podria considerar com una seqüència *stride* amb *delta* igual a 0. Aquestes seqüències són molt comunes, per exemple quan s'accedeix a un array d'una manera regular o en variables de bucles.

Les **seqüències no *stride*** comprenen la resta de seqüències que no pertanyen a cap de les dues classes anteriors. Aquestes seqüències apareixen quan es fan càlculs més complexos que no impliquen simplement afegir una constant. Per exemple moure's per una llista enllaçada produeix sovint valors que no tenen un patró *stride.*

Constant(C) 5 5 5 5 5 5 5...

Stride(S) 1 2 3 4 5 6 7 8...

Non-Stride(NS) 28 13 99 107 23 456...

Figura 10: Exemple de seqüències de dades

Així doncs l'especulació de dades es basa en què tenim algunes instruccions que escriuen de manera repetida el mateix valor a la seva adreça d'emmagatzemament, o el valor anterior més un increment constant (*strided values).* En el cas dels *loads* i els *stores* també s'observa el mateix fenomen, en alguns casos es tracta de la mateixa adreça, o la adreça anterior més un increment constant (reben el nom d'*strided references*).

En conseqüència de l'estudi de les seqüències anteriors han sorgit dos tipus de predictors de dades, els predictors computacionals i els basats en el context.

Els **Predictors computacionals** són els que fan una predicció mitjançant càlculs sobre valors previs que la instrucció ha generat. Bàsicament n'existeixen de dos tipus: el *Last Value Predictor* i el *Stride Predictor*.

El Last Value Predictor [7] és el tipus de predictor més simple. Es basa en la funció d'identitat: si l'últim valor produït per una instrucció és *v* la següent predicció serà *v.* S'acostuma a utilitzar un comptador associat al valor de manera que no es canvia la predicció fins que no s'hagi observat un cert nombre de vegades un valor diferent.

L'**Stride Predictor** [14] prediu el valor següent afegint al valor més recent la diferència entre els dos últims valors produïts per a la instrucció. La seva estructura podria consistir en una taula indexada pels *n* bits menys significatius del *Program Counter*, amb les següents entrades:

-Valor previ: El valor o la adreça de memòria de la última execució de la instrucció.

-Stride: Diferència de dos valors o adreces consecutives.

-Confidència: Un indicador que estableix com de bones són les prediccions sobre aquesta instrucció.

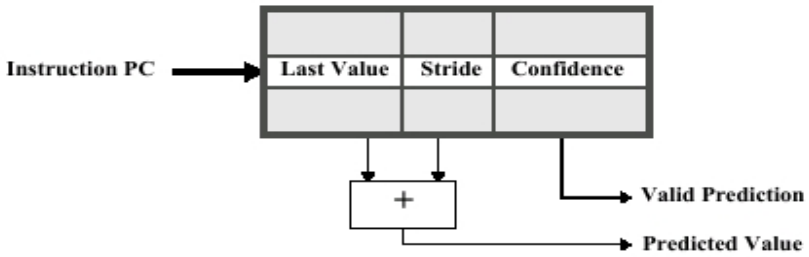


Figura 11: Esquema d'un Stride Predictor

A continuació es presenta una taula, a tall d'exemple, amb els resultats d'un predictor *Last Value* i un predictor *Stride* sobre diferents aplicacions dels SPEC95 [20] .

Benchmarks	Stride Predictor				Last Value Predictor			
	Result		Address		Result		address	
	pred (%)	mispred(%)	pred (%)	mispred(%)	pred (%)	mispred(%)	pred (%)	mispred(%)
go	38.4	3.3	26.8	4.7	30.4	4.5	25.6	4.0
m88ksim	54.8	0.6	42.0	4.6	42.0	2.7	31.2	1.3
ijpeg	16.7	0.9	19.4	2.2	17.4	4.4	18.1	2.2
perl	35.4	1.2	35.6	2.0	26.8	1.7	32.0	1.2
vortex	36.7	1.1	26.9	4.4	33.8	3.3	24.7	3.3
gcc	36.5	1.9	23.9	5.2	29.2	3.9	18.9	2.9
compress	20.5	0.2	43.4	0.03	17.3	0.6	41.7	0.1

Taula 2 : Exemple de taxes d'encerts dels predictors Last Value i Stride

Els percentatges de “resultat” són sobre el nombre total d'instruccions simulades, mentre que els percentatges de les adreces són sobre el nombre total de operacions de memòria. Com es pot observar, el nombre de valors i adreces que s'han pogut predir és significatiu (al voltant del 30% en mitjana) mentre que el percentatge de fallades en les prediccions és bastant petit en tots els casos. També es pot veure com el predictor *stride* és millor envers el predictor *last value*.

Els **Predictors basats en el context** [15]: aprenen els valors que segueixen un particular context i prediuen els valors quan el mateix context es repeteix. Això permet predir qualsevol seqüència que es repeteixi, sigui stride o no. Els més coneguts són els FCM (Finite Context Method Predictors), que es basen en predir el següent valor a partir de l'observació de k valors precedents. Els FCMs es construeixen mitjançant comptadors que compten les ocurrències d'un valor particular després d'un determinat context (patró). Així, per cada context ha d'existir, en general, tants comptadors com valors s'han trobat que segueixin el context en qüestió. El valor que es prediu és el que té el comptador amb el màxim valor.

Així doncs, malgrat que podria semblar que predir valors pot ésser una tasca difícil i limitada, està demostrat que les dades tendeixen a tenir **propietats de localitat** i en conseqüència un **alt nivell de predictibilitat**.

2.3.2.2. Especulació de dades mitjançant clústers

Tot i que per utilitzar l'especulació de dades mitjançant clústers es necessita la preparació explicita del codi, les seves característiques (com ara l'alta escalabilitat i el preu mòdic) els converteixen en

un entorn de treball idoni per l'especulació de dades.

Malgrat que pot semblar, i actualment pot ser, que el cost de comunicació i sincronització entre els diferents nodes d'un clúster fan que decaigui el guany que ens ofereix l'especulació, les lleis de Moore i de Glider fan pensar que amb el pas del temps aquest problema s'anirà alleujant.

D'una banda la llei de Moore, que és una llei empírica formulada per Gordon E. Moore l'any 1965, anuncia que aproximadament cada 18 mesos es duplica el nombre de transistors d'un processador. D'altra banda la llei de Glider apunta que la capacitat de les comunicacions es triplica cada any. Aquestes dues lleis estableixen que el cost de la transmissió d'informació i la sincronització entre les estacions de treball va fent-se menor enfront el cost de procés. Aquestes dues premisses porten a pensar que en el futur el temps de transport de les dades serà molt poc significatiu a l'hora d'aplicar tècniques d'especulació en un conjunt d'estacions de treball i per tant la idea de transportar aquestes tècniques a un entorn distribuït format per estacions de baix cost pot resultar molt rendible.

Per aquests motius dins del grup BCDS de la Universitat de Girona s'ha desenvolupat un prototipus, l'objectiu del qual és agafar un programa seqüencial com a entrada i executar-lo en un clúster d'estacions de treball aplicant tècniques d'especulació. Amb un sistema d'aquest tipus s'obtindria un rendiment millor al que es tindria amb una sola estació de treball, encara que el seu propòsit no és competir enfront als multiprocessadors o als processadors escalars, si no, disposar d'una metodologia que permeti explotar el paral·lelisme mitjançant clústers i minimitzar el temps de retorn dels programes.

Tot seguit parlarem del prototipus que hem esmentat.

2.4. Arquitectura Mestre/Esclau de Paral·lelització Especulativa per a Clústers

El grup de recerca BCDS de la Universitat de Girona ha proposat un esquema que permet aplicar tècniques d'especulació mitjançant clústers [10,11,12]. Aquest sistema consisteix en un software format per dos elements principals. El primer d'ells és el subsistema de paral·lelització que s'encarrega de transformar el codi original al format d'execució del motor d'especulació. El segon és el subsistema d'execució o motor d'especulació, que permet executar les aplicacions en paral·lel aplicant tècniques especulatives sobre un clúster, d'aquest ja es disposa d'una primera versió. A més a més, es disposa d'un simulador amb el propòsit de veure com funcionaria aquest model amb més nodes, i com afectaran, en els propers anys, els canvis de les velocitats dels processadors i del pas

de missatges descrits per la llei de Moore i de Glider.

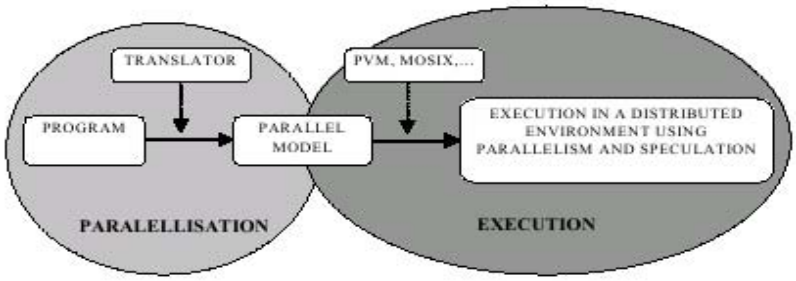


Figura 12: Esquema del sistema d'especulació

2.4.1. El subsistema de paral·lelització

El subsistema de paral·lelització s'encarrega de transformar una aplicació seqüencial en un format que pugui ser entès pel motor d'especulació. Aquest format ha de disposar del programa dividit en blocs que puguin ser executats en paral·lel. Per cadascun d'aquests blocs el conjunt de variables d'entrada i el de variables de sortida són coneguts.

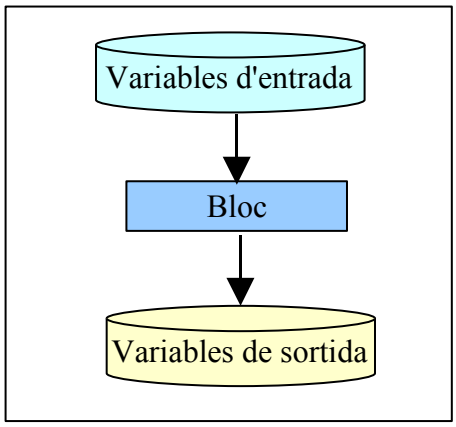


Figura 13: Bloc amb les seves dades d'entrada i de sortida

Concretament en aquest sistema es divideixen en blocs les estructures bàsiques com els bucles, condicions, funcions i procediments, tenint en compte que el criteri més important a l'hora de triar els blocs és el nombre d'instruccions que s'executaran en cadascun d'ells.

El resultat d'aquest subsistema són dos programes, el *Mestre* i l'*Esclau*. El Mestre és l'encarregat del paral·lelisme i de l'especulació del sistema, mentre que el codi de l'Esclau és el que s'executa en

cadascun dels nodes i anirà executant els diferents blocs en que s'ha dividit el programa seqüencial.

2.4.2. El subsistema d'execució

La sortida del sistema anterior, és la base d'aquest, el programa Mestre i l'Esclau. El programa Mestre és una part fixa per a totes les aplicacions i és qui dirigeix l'execució de l'aplicació, fet que inclou identificar quins són els blocs que estan preparats per a ser executats (tant sigui perquè les variables d'entrada ja estan calculades com perquè ja estan especulades), a més és l'encarregat d'assignar el node d'execució, de transmetre les dades d'entrada als esclaus, de recollir els resultats i anular les execucions errònies en el cas que sigui necessari.

Aquesta part intenta actuar com un processador superescalar, on les instruccions són els blocs en què el programa s'ha dividit, i els nodes del clúster que executen el codi esclau són l'equivalent a les unitats funcionals, i el node on s'executa el codi mestre seria la unitat de control.

El programa mestre permet que els blocs s'executin sense ordre i utilitza un sistema basat en l'algorisme **Tomasulo** per tal de trencar les dependències WAW i WAR tal i com ho faria un processador superescalar amb les instruccions. D'aquesta manera només ens queden les dependències RAW, que es veuen molt relaxades al aplicar tècniques d'especulació de dades. Per tal d'aplicar aquestes es disposa dels mecanismes **Last Value Predictor**, **Stride Predictor** i **Context-Based Predictor**. Pel que fa a les dependències de control, són manegades mitjançant la predicció de salts dinàmica, basant-se en el funcionament del **BTB** amb 2 bits d'historial [23]; en aquest cas no es necessita emmagatzemar l'adreça de destí, ja que es coneix quins són els possibles blocs a executar en qualsevol moment.

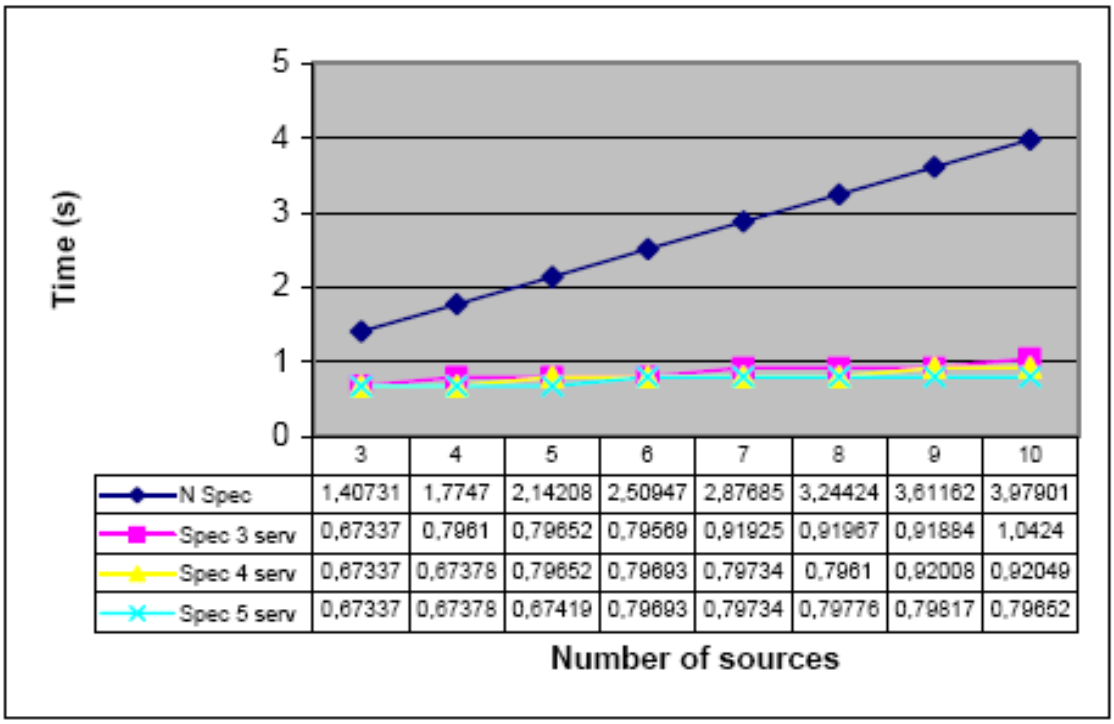
En el cas de fallar una predicció, només els blocs que s'han executat de manera incorrecta són descartats, i es continua l'execució des del darrer punt estable. Per tal d'aconseguir-ho s'utilitza una estructura basada en el **Reorder Buffer**.

En aquest punt, es veia com a possible coll d'ampolla l'execució del programa mestre per aquest motiu pot delegar les seves tasques a altres nodes, que alhora treballarien com a Mestres i s'anomenen nodes *Mestre/Esclau*. Aquest fet pot millorar el temps de retorn d'algunes estructures com poden ser els bucles niats.

La primera versió d'aquest subsistema està implementada amb C i PVM [13], i preparada per a ser executada en un clúster de màquines Linux amb PVM, encara que també pot ser executada en una sola estació de treball que actuï de mestre i d'esclau alhora. També cal dir que aquesta

implementació es va dissenyar per permetre més d'un fil d'execució sobre una mateixa dependència del graf, de manera que es pugui executar un mateix bloc amb diferents dades d'entrada simultàniament.

Com a exemple del rendiment que s'aconseguix, mostrem les simulacions d'un algorisme heurístic sobre el problema del viatjant de comerç (problema NP-hard), en què es disposa entre 3 i 5 nodes Mestre/Esclau i il·limitats nodes Esclau.



	3	4	5	6	7	8	9	10
N Spec	1,40731	1,7747	2,14208	2,50947	2,87685	3,24424	3,61162	3,97901
Spec 3 serv	0,67337	0,7961	0,79652	0,79569	0,91925	0,91967	0,91884	1,0424
Spec 4 serv	0,67337	0,67378	0,79652	0,79693	0,79734	0,7961	0,92008	0,92049
Spec 5 serv	0,67337	0,67378	0,67419	0,79693	0,79734	0,79776	0,79817	0,79652

Figura 14: Resultats obtinguts amb el problema del viatjant de comerç

Com es pot observar l'execució mitjançant especulació en múltiples computadors ha reduït el temps de manera considerable sobre l'execució sense especulació. També cal esmentar que es tracta d'un exemple que permet predir amb facilitat els valors de les variables i incrementar de manera significant el grau de paral·lelisme del programa. Pel que fa a l'aplicació d'un major nombre de nodes Mestre/Esclau, cal fixar-se concretament en els valors temporals per veure'n una diferència.

Incrementar el nombre de Mestres/Esclaus només afavorirà en aquells casos on el resultat de la divisió del nombre de ciutats entre els nodes Mestres/Esclaus arrodonits cap amunt disminueixi.

Les etapes d'execució d'un bloc són la gestió inicial, l'enviament del missatge amb el número de bloc a executar i les dades d'entrada, l'execució del bloc, l'enviament del missatge amb la

informació del bloc executat i les dades de sortida i la gestió per finalitzar el bloc.

Gestió Inicial	Missatge Inicial	Execució	Missatge Final	Gestió Final

Figura 15: Etapes d'un bloc

L'etapa de gestió inicial, el node Mestre prepara tota la informació necessària per executar el bloc i l'envia al node Esclau o Mestre/Esclau encarregat d'executar-la, que quan rep el missatge reconèix el bloc a tractar, l'executa prepara les dades de sortida i les envia al node Mestre que actualitzarà les dades.

En el cas que el missatge vagi destinat a un node Mestre/Esclau, l'etapa d'execució consistirà en totes les etapes d'un conjunt de blocs, encara que el node Mestre ho veurà com un bloc qualsevol i el node Mestre/Esclau serà l'encarregat de gestionar aquests blocs.

2.4.3. El simulador

Un dels problemes d'aquest entorn, és la dificultat de disposar d'un clúster amb un nombre de nodes realment elevat, tot i que el cost dels PC's sigui molt assequible avui en dia.

Per aquest motiu s'ha desenvolupat un simulador que utilitza les dades sobre els diferents costos temporals de l'execució i permet obtenir el temps que trigaria l'execució si es disposessin de més nodes.

El simulador utilitza els següents temps:

- Temps d'*start up* (Tg1): és el temps necessari per comprovar si es pot llançar un procés i obtenir les seves dades d'entrada.
- Temps de transmissió d'un missatge (Tt): és el temps per enviar i rebre un missatge entre dos processos.
- Temps d'actualització de dades (Tg2): és la mitjana de temps que necessita el master per llegir i emmagatzemar les dades de sortida dels blocs.
- Temps d'eliminació d'un procés (Tm): és el temps necessari per eliminar les sortides dels processos on s'ha fallat l'especulació.

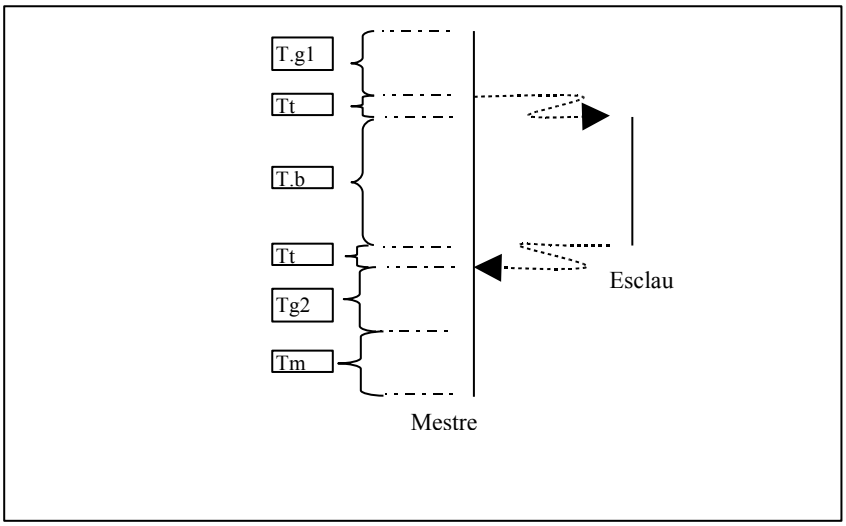


Figura 16: Esquema dels diferents temps d'un bloc

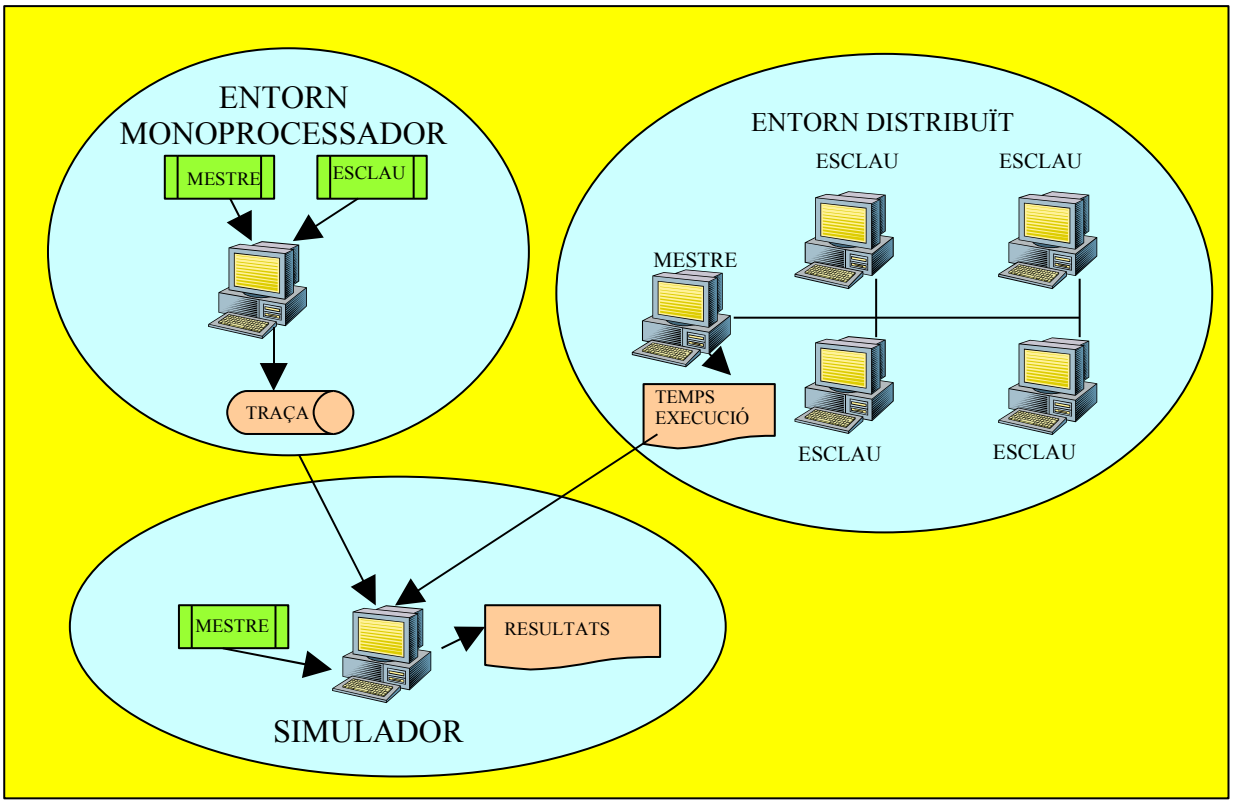


Figura 17: Esquema de l'estructura d'ús del simulador

El simulador però, està pensat perquè els blocs del mateix tipus tinguin un cost molt similar, ja que se n'utilitza una mitjana, tampoc està preparat pel model Mestre-Mestre/Esclau-Esclau i la seva adaptació es preveu força complicada.

2.4.4. Model analític

Un programa seqüencial es divideix en blocs, de manera que la seva execució consisteix en la de B blocs. Els blocs necessiten de mitjana un temps d'execució Tb, un de transmissió Tt tant per les dades d'entrada com de sortida, un de gestió Tg (Tg1+Tg2) per part del Mestre (entre inicialitzar i finalitzar el bloc). Si suposem una especulació perfecta que pot predir les dades d'entrada i l'especulació de control sense error, el temps mitjà d'execució per bloc (Tbm) serà el següent:

$$\text{Tbm} = \begin{cases} \dfrac{Tg+Tb+2*Tt}{N} & \text{si} \quad \dfrac{Tg+Tb+2*Tt}{Tg} > N \\ Tg & \text{altrament} \end{cases}$$

Equació 5: Temps mitjà d'execució per bloc

2.5. Introducció a l'Algorisme Fast Multipath Radiosity Using Hierarchical Subscenes

En aquest apartat explicarem el funcionament de l'algorisme *Fast Multipath Radiosity Using Hierarchical Subscenes* [2,3,4], primer de tot, però, començarem explicant les tècniques de radiositat i l'algorisme del Multipath.

2.5.1 Radiositat i factors de forma

Les tècniques de radiositat estimen la il·luminació d'una escena amb superfícies difuses, que són aquelles que reflecteixen la llum en totes direccions i amb la mateixa intensitat. Prèviament l'escena és discretitzada en elements que s'anomenen *patxos*, per cadascun d'ells la radiositat, la reflectància i l'emitància (veure més avall) es consideraran constants. Per a cada patx podem calcular la seva radiositat amb la següent equació:

$$B_i = E_i + \rho_i \sum_j F_{ij} B_j$$

Equació 6: Equació discreta de la radiositat per a cada patx *i*.

on:

- B_i és la radiositat del patx *i*.

- E_i és l'emitància del patx *i*, és a dir la potència emissora per unitat de superfície que emet com a font primària el patx *i*.

- ρ_i és la reflectància del patx *i*, és a dir la proporció de llum que reflecteix el patx.

- F_{ij} és el factor de forma del patx *i* al patx *j*, que és la fracció de llum que va del patx *i* al *j* de manera directa.

Si s'aplica a tots els patxos s'obté un sistema d'*N* equacions lineals, on *N* és el nombre de patxos. La part més costosa de computar són els factors de forma que representen la visibilitat entre els patxos.

El factor de forma F_{ij} és la fracció de l'energia que surt del patx *i* que va directament al patx *j*. La seva equació es pot expressar de la següent manera:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \left(\frac{\cos(\theta_i)\cos(\theta_j)}{\pi r^2_{(x_i, x_j)}} \right) V(x_i, x_j) \delta A_j \delta A_i$$

Equació 7: Form Factor del polígon *i* al polígon *j* .

on:

- A_i és l'àrea del patx *i*.

- θ_i, θ_j és l'angle entre la normal del patx *i* o *j* respectivament i la línia que uneix ambdós patxos.

- $V(x_i,x_j)$ és la funció binària de visibilitat entre els punts x_i i x_j.

- $r(x_i,x_j)$ és la distància entre els els punts x_i i x_j.

Cal dir que els factors de forma depenen únicament de la geometria de l'escena. El sistema es pot resoldre calculant primer els factors de forma (té un cost $O(n^2)$ en memòria, on *n* és els nombre de patxos), o bé sense calcular-los explícitament.

Els mètodes de Monte Carlo són mètodes probabilístics que resolen problemes matemàtics mitjançant la simulació de variables aleatòries. Aquests mètodes es poden utilitzar per estimar integrals de les quals no es pot trobar una solució analítica (rep el nom d'integració de Monte Carlo). Els mètodes de Monte Carlo es poden utilitzar per resoldre els factors de forma, tant explícitament com implícitament, mitjançant el traçat de línies aleatòries.

Les línies aleatòries es poden generar amb un enfocament local o global. En el local, les línies s'originen en un patx i finalitzen en la primera intersecció: Aquestes línies reben el nom de línies locals i la seva simulació el de Monte Carlo Local. En el global, les línies van d'un extrem a l'altre de l'escena i es consideren totes les interseccions; reben el nom de línies globals i la seva simulació el de Monte Carlo Global.

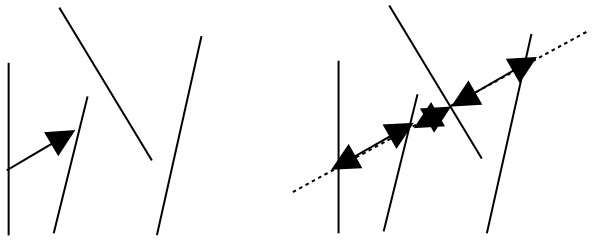


Figura 18: A l'esquerra hi tenim una línia local, mentre que a la dreta una de global

2.5.2. El mètode multipath

El mètode de multipath [16] és un algorisme de radiositat que utilitza línies globals per a la simulació de la trajectòria dels fotons. Les línies globals serviran per simular l'intercanvi energètic entre parelles de patxos, de manera que cada línia contribuirà a diversos camins de les partícules de llum[4]. Els factors de forma no es calculen explícitament, sinó que són simulats amb els segments de les línies globals.

Considerem la versió de *shooting* del multipath, en la qual la idea és disparar potència de les fonts de llum. Cal dir que existeix una versió de *gathering*, que en general i sense entrar en detalls, no resulta tan eficient.

2.5.2.1. L'algorisme multipath

Les línies globals intersequen tota l'escena (suposem un entorn tancat). Per a cada línia les interseccions s'ordenen per distància. Per cada patx es guarden dos valors, la potència acumulada i la potència unshot (pendent de disparar). Cada parella de patxos de la llista d'interseccions intercanvia la seva potència unshot disminuïda per la reflectància. A més, la potència unshot de cada patx de la parella s'afegeix a la potència acumulada de l'altre patx, decrementada també per la seva reflectància. Si un patx és a més una font lumínica, caldrà afegir la seva potència emissora per línia, que es calcula a priori dividint la potència de cada font de llum pel nombre previst de línies que creuaran la font, el qual, per geometria integral [17], s'estima com a:

[4] D'aquí surt el nom del mètode de multipath.

$$N_{\text{inter}_i} = \text{N} * \frac{2 * A_i}{A_{o_i}}$$

Equació 8: Nombre d'interseccions previstes per al patch i

on A_{o_i} és l'àrea de la capsa o esfera des d'on es generen les rectes i *N* el nombre de línies que s'hi llancen. De manera que la potència per raig és:

$$ppr_i = \frac{\Phi_i * A_{o_i}}{N * 2 * A_i} = \frac{A_{o_i}}{2 * N} * E_i$$

Equació 9: Potència per raig del patx *i*

La següent figura conté l'algorisme del mètode, on les variables emprades són: ρ_i que ens indica la reflectància del patx *i*, B_i que conté l'estimació de la radiositat del patx *i*, pow_i_j que indica la potència transportada del patx *i* al *j*, accum_i que conté el total acumulat de potència reflectida pel patx *i*, unshot_i que emmagatzema la potència del patx *i* que s'ha de transmetre a altres patxos, E_i que indica la potència la potència primària del patx *i*, ppr_i que conté la potència primària per línia del patx *i* i A_i que conté la àrea de *i*.

```
algorisme Multipath
      Carregar l'escena
      Subdividir els polígons en patxos
      per cada patx i
            accum_i=0, unshot_i=0, emit_i=0, ppr_i=0
      fiper
      per cada font de llum i
            inicialitzar ppr_i
      fiper
      per k=1 fins N
            Generar línia k
            Calcular la llista d'interseccions ordenades de la línia k
            per cada parella de patxos i , j de la llista
                  pow_i_j= (unshot_i + ppr_i) * ρ_j
                  powj_i = (unshot_j +ppr_j) * ρ_i
                  unshot_i= pow_j_i
                  unshot_j=pow_i_j
                  accum_i=accum_i + pow_j_i
                  accum_j=accum_j + pow_i_j
            fiper
      fiper
      per cada patx i
            B_i=(accum_i +E_i)/A_i
      fiper
      Escriure l'escena
fialgorisme
```

Figura 19: Algorisme del Multipath

Un avantatge d'aquest mètode és que totes les interseccions són utilitzades i la transferència de potència és bidireccional cosa que fa que cada línia contribueixi a més d'un camí a l'hora de simular la trajectòria dels fotons. D'altra banda, a les primeres fases la potència només pot provenir de les fonts de llum, de manera que les línies que no hi intersequen, no aporten cap simulació de la trajectòria dels fotons, i són computades en va. Aquest desavantatge, però, es pot resoldre aplicant un procés anomenat first shot, amb el propòsit de repartir l'energia directa provinent de les fonts de llum mitjançant línies locals, de manera que la distribució de la potència emissora sigui més uniforme.

2.5.2.2 Fast multipath radiosity using hierarchical subscenes

En aquest apartat comentarem les millores que inclou l'algorisme *fast multipath radiosity using hierarchical subscenes* [2,3,4] respecte l'anterior. La idea principal és treballar amb una jerarquia de subescenes, de manera que quan considerem una subescena no ens preocupem ni de l'exterior ni de les subescenes interiors (Figura 20).

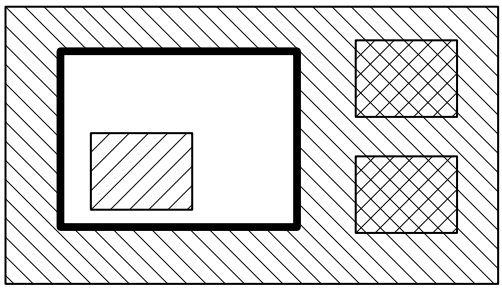


Figura 20: La subescena considerada és la remarcada en negre. Quan sigui processada, només es consideraran les interseccions de la part blanca.

L'algorisme (Figura 21), divideix l'escena a tractar en una jerarquia de subescenes. Cada subescena serà envoltada d'una caixa virtual i les seves parets seran discretitzades en patxos virtuals, i l'hemisferi sobre cada patx virtual se subdivideix en regions angulars. Tot seguit, hi ha un preprocés en què es llancen línies globals per a tota l'escena i també per a cada subescena. En cada cas, es tindran en compte les interseccions amb els seus polígons, amb els patxos virtuals de les subescenes internes i amb les parets virtuals de la pròpia subescena (o bé les parets reals, si es tracta de l'escena principal). A més a més, en aquest procés s'estimaran les transmitàncies (veure més endavant).

```
algorisme FMR
            Carregar l'escena
            Subdividir els polígons en patxos
```

```
            Generar la jerarquia de subescenes
            Subdividir cada caixa virtual en patxos virtuals i regions angulars
            // inici preprocés
            per cada subescena S (inclosa l'escena sencera)
                  Llançar línies globals i emmagatzemar les llistes
                  d'interseccions
                  si S no és l'escena sencera
                        per cada regió angular de S
                              computar les transmitàncies
                        fiper
                  fisi
            fiper
            // fi preprocés
            First shot
            per cada iteració
                  MP(escena sencera)
            fiper
            Escriure l'escena
fialgorisme
```

Figura 21: Algorisme

Tot seguit explicarem amb més detall les diferents parts de l'algorisme.

Generar la jerarquia de subescenes

Per tal de generar la jerarquia de subescenes, es poden usar moltes estratègies diferents. Que es poden basar per exemple en heurístiques que afecten els quocients d'àrees de les capses, etc.

Subdividir cada caixa virtual en patxos virtuals i regions angulars

Les parets virtuals es discretitzen en patxos virtuals de la mateixa manera que els polígons en patxos. A més, l'hemisfera de direccions sobre cada patx virtual se subdivideix en regions angulars (Figura 22). Les regions angulars actuaran com a acumuladors d'energia entrant i sortint a la capsa.

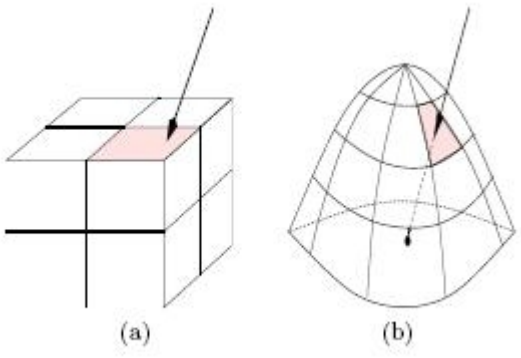


Figura 22: La figura (a) mostra la subdivisió en patxos virtuals de les parets de les caixes virtuals, mentre que a la (b) podem observar la subdivisió en regions angulars dels patxos virtuals.

Preprocés: llançar les línies globals

Per cada línia global que es llança en una subescena S, se n'ha de calcular la llista d'interseccions, i ordenar-les per distància. S'han de contemplar les interseccions amb els objectes de S, amb les regions angulars de les subescenes que S contingui i amb les seves parets virtuals (o bé amb les parets físiques si es tracta de la caixa principal). Pel que fa a les subescenes de S, no es necessari calcular-ne les interseccions internes. (Figura 23)

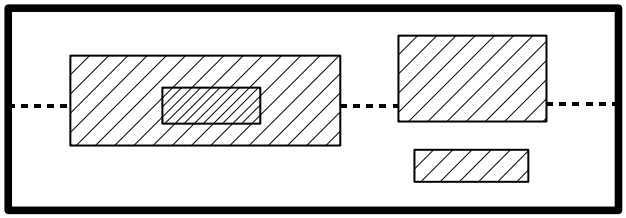


Figura 23: La línia puntejada representa una línia global per a la subescena remarcada en negre, les caixes ratllades són subescenes internes de S, l'interior de les quals no serà considerat quan generem rectes per a S.

Per a cada escena i subescena, s'emmagatzema al disc dur la llista de les interseccions de cada línia global ordenades per distància. Per a cada intersecció es guarda un identificador que permet distinguir la regió angular o patx intersecat.

El nombre de línies que s'ha de llançar a cada subescena es determina de forma heurística, com pot ser repartir les línies en proporció al nombre d'objectes que conté la subescena, etc.

Càlcul de la transmitància de cada regió angular

La transmitància d'una regió angular R representa la proporció de la potència entrant a la subescena per les direccions associades a R que travessa la capsa. Aquesta mesura ens proporciona una estimació de la opacitat d'una subescena per cada regió angular, i es pot aproximar de la següent manera:

$$T_R \approx \frac{n_R^d}{n_R}$$

Equació 10: Transmitància de la regió angular R

on T_R és la transmitància de la regió angular R, n_R^d és el nombre de línies corresponents a R i que travessen la subescena sense haver trobat cap obstacle, i n_R és el total de línies corresponents a R. Com més proper a 0 sigui T_R més opaca serà la subescena en les direccions associades a R.

Les transmitàncies permeten que s'ignori el contingut de les subescenes amb una pèrdua de precisió

moderada.

First shot

Com s'ha comentat anteriorment, el first shot s'encarrega de llançar línies locals des de les fonts de llum amb l'objectiu d'expandir-ne la potència lumínica. . El nombre de línies que s'originarà a cada font serà proporcional a la seva potència.

Acció Multipath

En el procés iteratiu la potència que surt i arriba de les subescenes s'acumula en les regions angulars, de manera que és necessari que hi hagi intercanvi energètic entre elles. Abans de cada iteració, cal establir la potència corresponent a cada línia entrant i sortint per cada una de les regions.

En haver llançat inicialment les línies globals, en tot moment sabem quantes han intersecat cada cada regió angular, i per tant no s'han de predir com passava a l'algorisme original. Les potències per línia s'estableixen com s'indica a les següents eqüacions:

$$\text{pple}_R = \frac{\text{Potència Entrant Acumulada}_R}{\text{Nombre d'Interseccions de línies entrants}_R}$$

Equació 11: potència per línia entrant a l'escena per la regió R

$$\text{ppls}_R = \frac{\text{Potència Sortint Acumulada}_R}{\text{Nombre d'Interseccions de línies de sortida}_R}$$

Equació 12: potència per línia sortint a l'escena per la regió R

```
acció Multipath (escena S)
      Calcular pplp_P de cada patx de S que no formi part de cap de les seves
      subescenes
      si S ≠ escena principal
            per cada Regió Angular R de S
                  Calcular pple_R
                  Posar a 0 la potència sortint de R
            fiper
      fisi
      per cada subescena S_i de S
            per cada Regió Angular R de S_i
                  Calcular ppls_R
                  Posar a 0 la potència entrant a R
            fiper
      fiper
      per cada línea llançada a S
            Llegir la llista d'interseccions L calculada al preprocés
            Intercanvi_energètic(L)
      fiper
```

```
      per cada subescena S_i de S
            Multipath(S_i)
      fiper
fiacció
```

Figura 24: Algorisme Multipath Recursiu

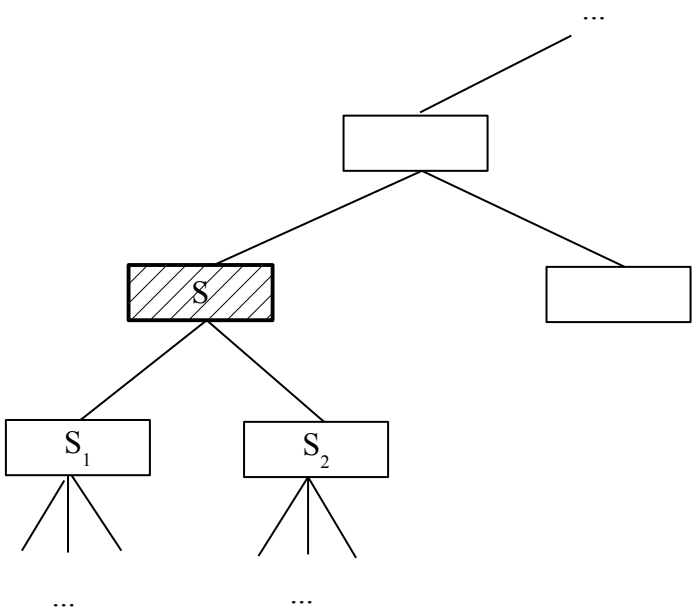


Figura 25: Representació de la jerarquia de subescenes.

```
algorisme Intercanvi_Energètic(Línia L)
      Obtenir la llista d'interseccions de L
      Intercanviar energia entre les parelles d'interseccions, de la següent
      manera:
      * Cada patx envia la seva potència unshot i la seva ppr
      * Les regions angulars de les subescenes de S envien la seva ppls
      * Les regions angulars de S envien la seva pple
      * Cada patx actualitza la seva potència unshot amb la rebuda, i incrementa
        la seva potència acumulada
      * Les regions angulars de les subescenes de S incrementen la seva potència
        entrant acumulada i la sortint (amb la pple de la seva regió angular
        oposada multiplicada per la transmitància)
      * Les regions angulars de S incrementen la seva potència sortint acumulada


fialgorisme
```

Figura 26: Algorisme Intercanvi energètic entre les interseccions d'una Línia L

3. Metodologia

La tasca a desenvolupar la hem dividit en les següents fases:

1. **Implementació de l'algorisme**: l'hem implementat en C, sense utilitzar punters a memòria ni recursivitat, i a més a més procurant de fer un codi el màxim de linial possible.
2. **Generar els blocs pel model Mestre-Esclau**: hem dividit la implementació en blocs i hem creat el graf de dependències pel model Mestre-Esclau
3. **Execució concurrent**: hem modificat el motor d'especulació per executar la implementació concurrentment mitjançant pipes.
4. **Paral·lelitzar la implementació amb PVM**: un cop hem vist que la versió concurrent funciona, la convertim a paral·lela utilitzant PVM
5. **Generar els blocs pel Model Mestre-Mestre/Esclau-Esclau**: per tal d'augmentar més el grau de paral·lelisme hem dividit aquells blocs que són paral·lelitzables per tractar-los amb el nou model
6. **Implementació amb model Mestre-Mestre/Esclau-Esclau amb PVM**: un cop els blocs han quedat dividits cal modificar el motor d'especulació.
7. **Optimització del codi**: en aquest punt hem intentat millorar el rendiment obtingut com també paral·lelitzar més algun bloc.
8. **Adaptar el simulador**: el simulador que disposàvem tenia algunes limitacions que han calgut resoldre.
9. **Obtenir resultats**: hem utilitzat el temps de resposta en execucions reals en el simulador per tal d'obtenir-ne amb més nombre de nodes i aplicant les lleis de Moore i de Glider.

4. Part experimental

En aquest apartat explicarem els treball desenvolupat a cada fase del projecte, seguint l'esquema indicat a l'apartat de la metodologia.

4.1. Implementació Algorisme

Inicialment disposàvem d'una implementació de l'algorisme *Fast Multipath Radiosity Using Hierarchical Subscenes,* programada amb C++, que utilitza memòria dinàmica i alguns algorismes recursius. Cosa que introdueix dificultat a l'hora de paral·lelitzar-lo amb el prototipus esmentat, ja que la posició en memòria d'un mateix objecte variaria en cada node i que els algorismes recursius creen moltes dependències de dades i limiten l'execució en paral·lel. Per aquest motiu ha calgut tornar a programar l'algorisme, i ho hem fet procurant obtenir un codi el més seqüencial possible per tal d'aconseguir més flexibilitat a l'hora de dividir-lo en blocs. Cal dir que tot i que la implementació original estava programada amb orientació a objectes, no feia us de l'herència, fet que hauria pogut dificultar molt més aquest procés.

En aquesta fase ens ha sigut necessari estudiar la implementació que disposàvem per tal d'obtenir i entendre l'algorisme específic de la implementació.

Les dues implementacions parteixen d'un fitxer de l'escena amb format MGF[5][8], on es descriuren els objectes i els seus materials, per generar un fitxer VRML[6][22] amb l'escena il·luminada.

En aquest apartat parlarem principalment de l'adaptació de les estructures de dades i com s'ha eliminat la recursivitat dels algorismes recursius.

4.1.1. Estructures de dades i eliminació de la recursivitat

A la implementació disposem de diverses taules on contenim els diferents tipus d'estructures. Al no usar estructures dinàmiques, els nodes contenen dos enters, el primer indica la posició dels seus nodes fills, que es crearan a la primera posició lliure de la taula, i el segon el nombre de fills que tenen. Les principals estructures i els atributs que els hi hem definit són les següents:

[5] Materials and Geometry Format
[6] Virtual Reality Modeling Language

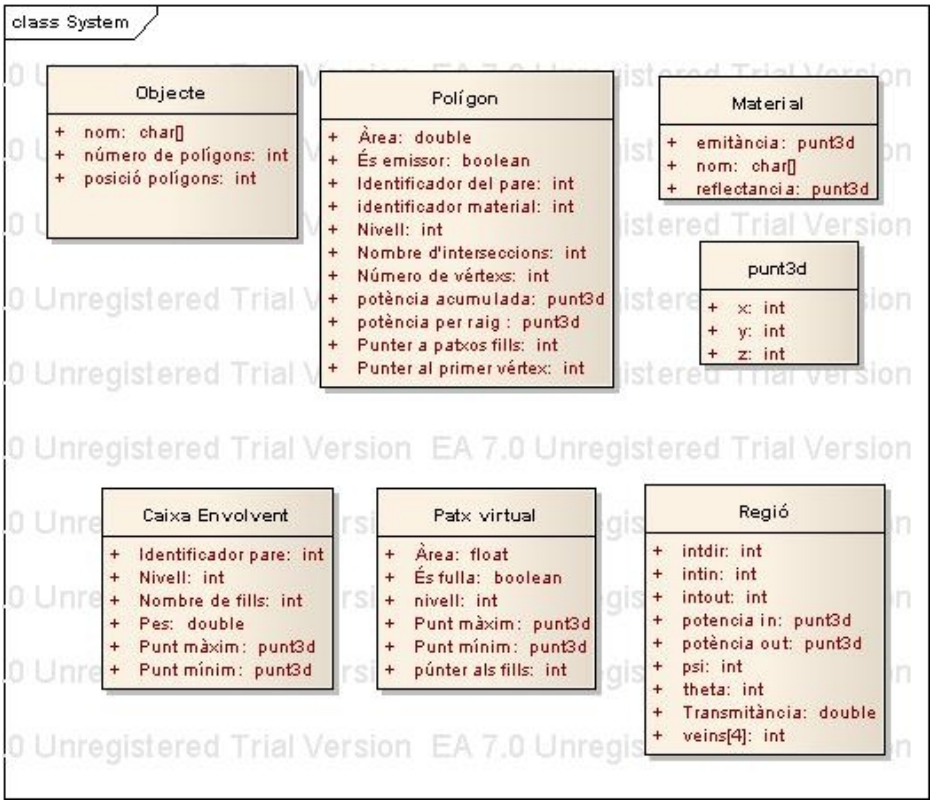


Figura 27: Estructures de dades i els seus atributs

Pel que fa a la recursivitat, com que en la majoria dels casos consistia en tractar els nodes fills després dels pares, la hem pogut resoldre tractant les taules per ordre, ja que els nodes pares sempre estan situats abans que els fills. A les següents figures mostrem com a exemple els algorismes originals i els nous de la discretització dels polígons:

```
algorisme Discretització de polígons
      N=nombre de polígons
      per i=1 fins a N
            si polígon[i].Àrea>=llindar
                  Subdividir(polígon[i])
            fsi
      fiper
fialgorisme

algorisme Subdividir(polígon N)
      per i=1 fins 4
            N.fill[i]=N.divisió(i);
            si N.fill[i].Àrea>=llindar
                  Subdividir(N.fill[i])
            fsi
      fiper
fialgorisme
```
Figura 28: Algorisme recursiu

```
algorisme Discretització de polígons
      N=nombre de polígons inicials
      i=1
      mentre i<=N
            si polígon[i].Àrea>=llindar
                  polígon[i].punter_fills<-N
                  per j=1 fins a 4
                        polígon[N]=polígon[i].divisió(j)
                        N=N+1
                        polígon[i].Nombre_fills=polígon[i].Nombre_fills+1
                  fiper
            fisi
            i=i+1
      fimentre
      nombre de polígons finals = N
fialgorisme
```

Figura 29: Algorisme iteratiu

Pel que fa a la divisió de les parets virtuals, com que cada paret sempre es divideix en patxos virtuals de la mateixa àrea, hem vist que podíem calcular el nombre de divisions necessàries i dividir-la-hi directament, enlloc de fer-ho recursivament generant resultats intermitjos que caldira emmagatzemar, com és el cas de la discretització dels polígons.

$$\text{Nivell} = \left\lceil \frac{\log\left(\frac{A}{A_{\max}}\right)}{\log(4)} \right\rceil \qquad NPV = 4^{Nivell}$$

Equacions 13 i 14: El nivell fa referència al nivell de l'arbre que contindria els patxos virtuals. NPV és el nombre de patxos virtuals que s'han de generar.

Tot seguit mostrem unes imatges d'una escena VRML generada amb la nova implementació.

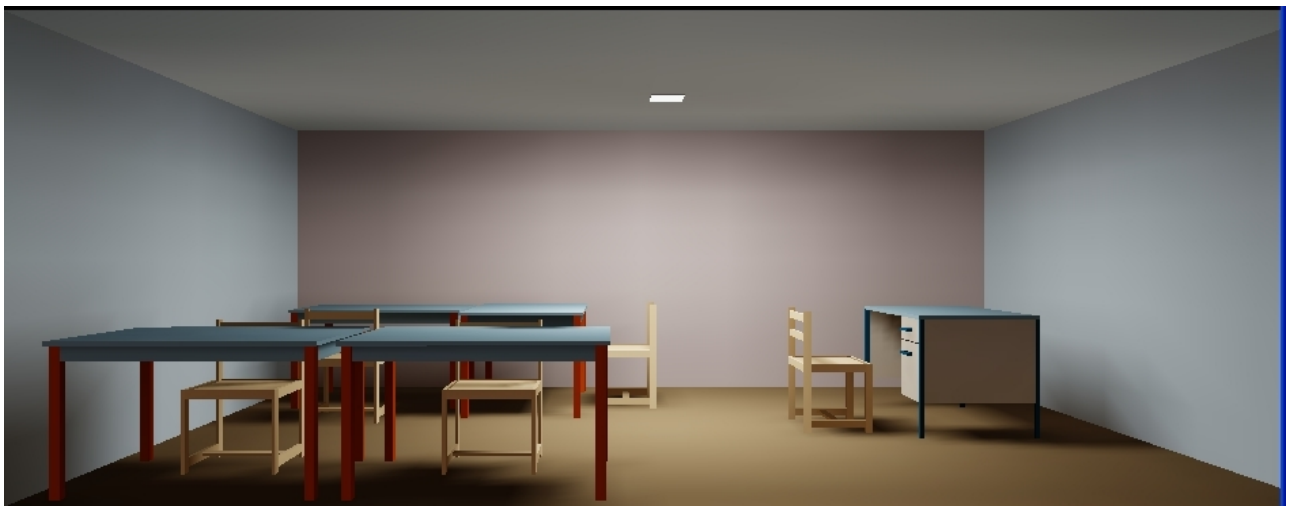


Figura 30: Imatge d'una escena resultant de l'algorisme implementat.

Figura 31: Imatge d'una escena resultant de l'algorisme implementat, 2.

4.2. Generació de blocs pel model Mestre-Esclau

El motor d'especulació necessita que l'algorisme a executar es divideixi en blocs bàsics. En el nostre cas l'hem dividit en els que mostren les següents figures, en les quals també es pot veure les dades que consumeixen i produeixen.

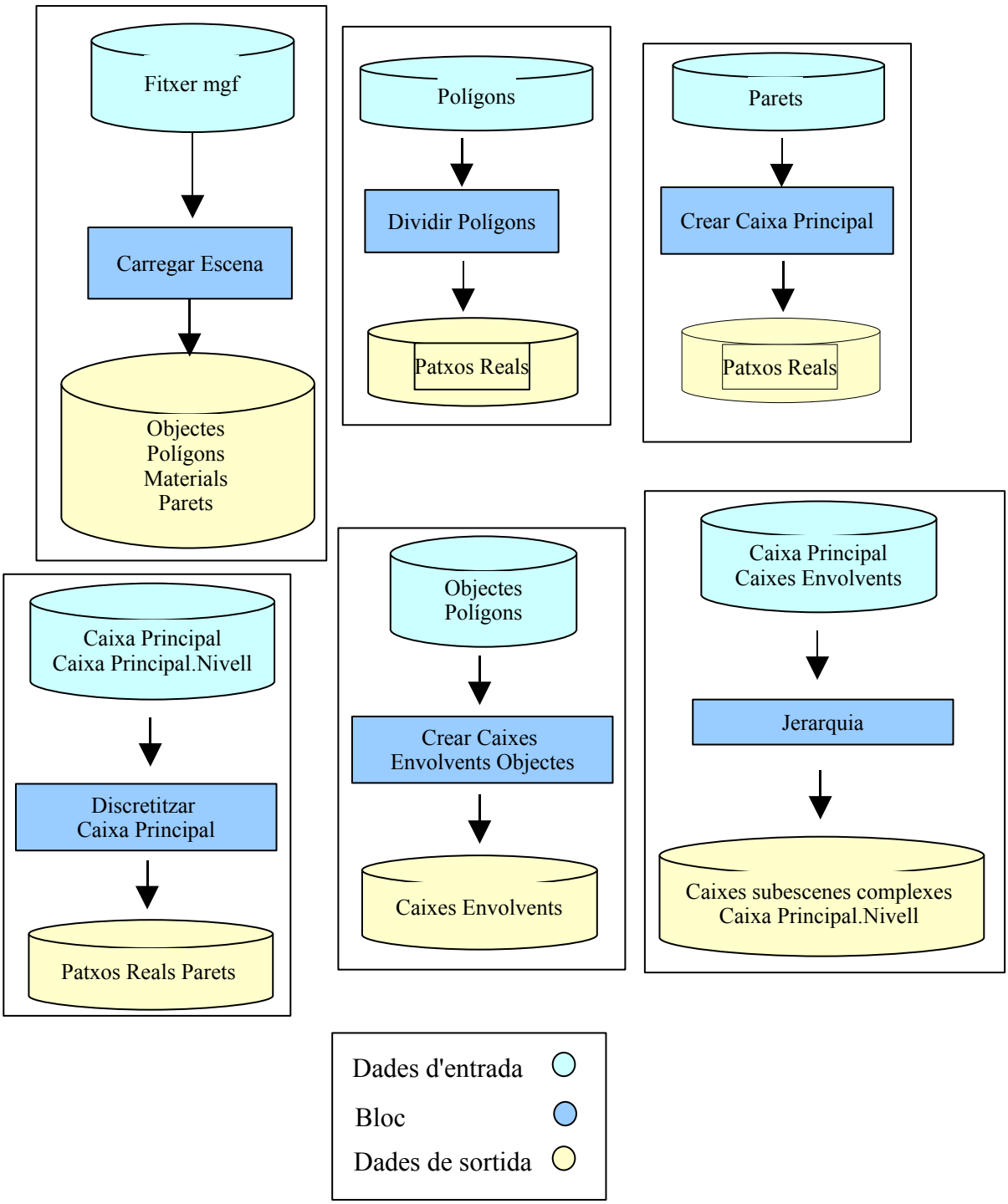


Figura 32: Blocs amb les dades consumides i produïdes, part 1.

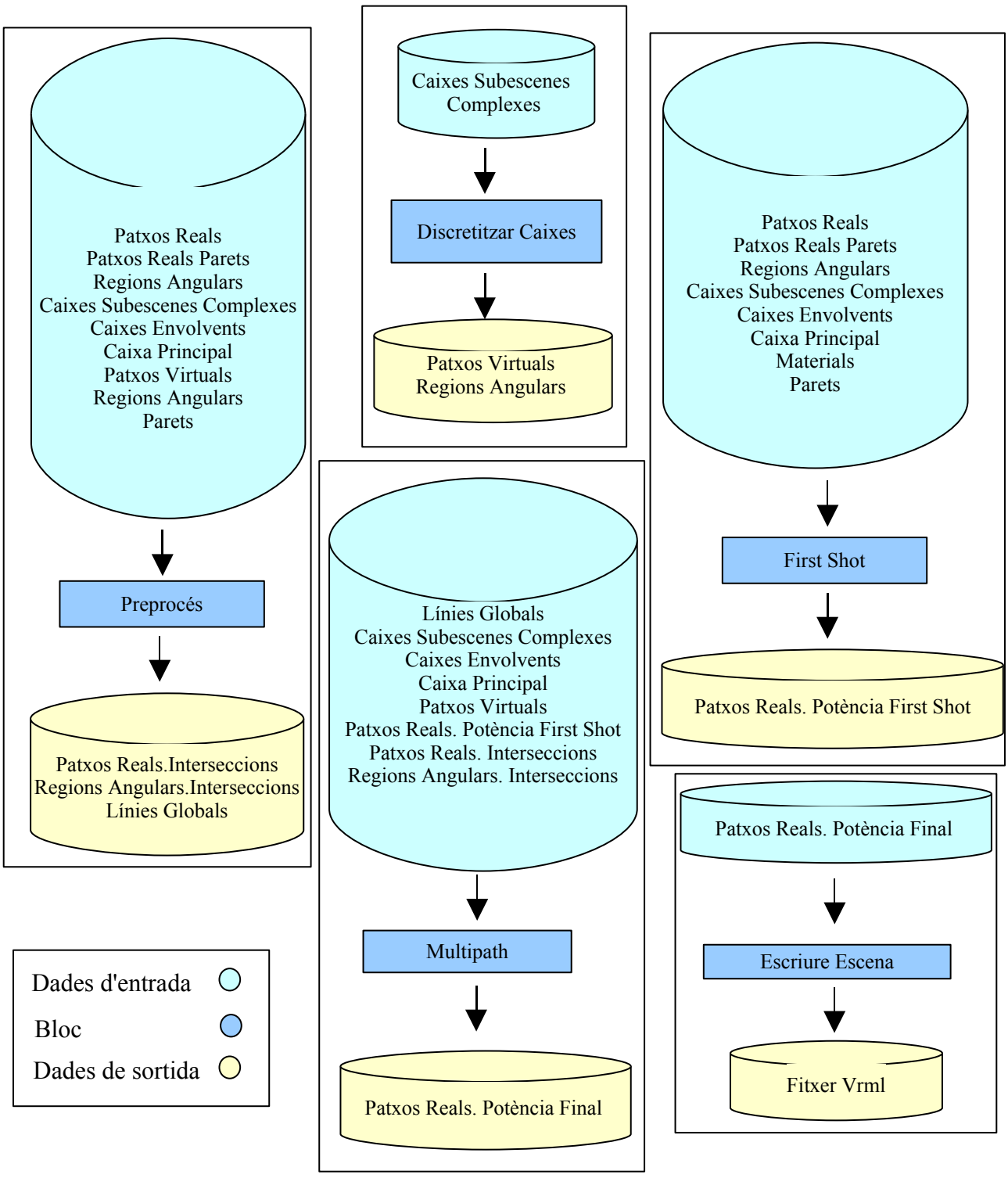


Figura 33: Blocs amb les dades consumides i produïdes, part 2.

En el nostre cas, l'algorisme de gràfics ha quedat dividit en els següents blocs i en el següent graf de dependències des del punt de vista del node Mestre:

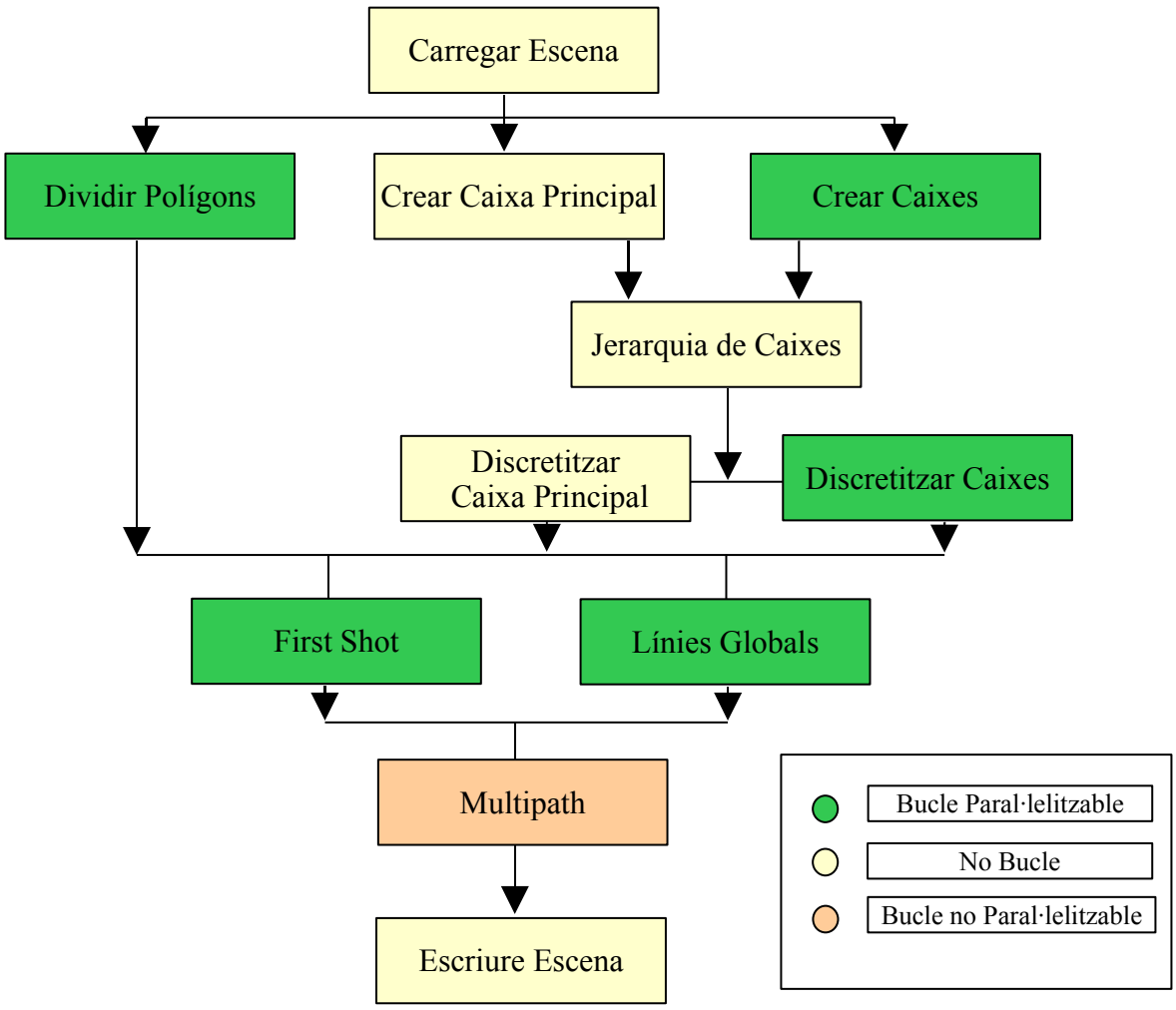


Figura 34: Graf de dependències entre blocs.

Com a bucle paral·lelitzable entenem aquells que es poden executar alhora mitjançant l'especulació de la variable d'inducció, i com a bucle no paral·lelitzable aquells que encara que es pugui predir la variable d'inducció, tenen altres dependències que no són especulables.

Tot i que el First Shot no depèn del bloc *Discretitzar Caixes,* com que es tracta d'un bloc d'una execució curta hem decidit lligar-lo per facilitar la gestió, cosa que comentarem al següent apartat.

Pel node Mestre s'executa la discretització de polígons una vegada per polígon, en la creació de caixes i una vegada per objecte, en la discretització de subescenes tenim un bloc per cada una

d'elles, en el first shot per cada patx emissor de llum, i en el preprocés per l'escena principal i per a cada subescena. El Multipath s'executa per cada iteració que s'ha demanat, però com hem vist anteriorment hi ha dependències entre iteracions.

Cal dir que en fer aquesta distribució, el node Mestre és l'encarregat d'ajuntar els resultats dels diferents nodes.

4.3. Execució concurrent

En aquest apartat explicarem les modificacions que hem fet a la implementació del subsistema d'execució per tal d'executar l'algorisme de gràfics de manera concurrent.

Cal dir que el motor d'especulació, al tractar-se d'un prototipus, té una sèrie de limitacions que fan que l'haguem d'adaptar per tal de paral·lelitzar l'algorisme *Fast Multipath Radiosity Using Hierarchical Subscenes*. En primer lloc la mida dels missatges entre els diferents nodes era constant, cosa que comportava un problema de memòria per la quantitat de dades que s'envien en alguns dels mètodes. En segon lloc, si es vol que els blocs d'una estructura for tinguin totes les dades actualitzades, es poden crear una gran quantitat de dependències que impossibiliten l'encert de la predicció de totes elles, de manera que cal mirar de relaxar el tractament d'algunes variables fent que el valor no s'actualitzi fins el final del bucle. A més, ens hem trobat amb algun problema imprevist amb l'ús de les pipes per la comunicació.

Inicialment els missatges que s'enviaven d'un node a un altre contenien una estructura estàtica amb tota la informació necessària. Enlloc d'augmentar la mida d'aquesta estructura, el que s'ha fet és enviar a continuació la resta d'informació, perquè la informació s'hi escrigui de cop, primer la hem emmagatzemat en un buffer des del que hem escrit a la pipe. Per fer-ne el tractament hem modificat el mètode de gestió d'inicialització i finalització dels blocs, adaptant-los al tipus de dades que es rebien i al tractament que calia fer en rebre el resultat.

En aquesta versió del motor d'especulació s'utilitzen dues pipes per comunicar els processos. Una pipes és d'escriptura pel Mestre i lectura pels Esclaus i l'altra a l'inrevés, de manera que els Esclaus comparteixen l'accés a les pipes. El problema que no hi havia cap control sobre l'accés a les pipes i en la versió de partida no suposava cap incovenient ja que la mida de la informació que s'hi escrivia era suficientment petit, perquè la lectura i escriptura es realitzes de manera atòmica. Ara bé, quan els missatges han augmentat de mida, les operacions de les pipes han deixat de ser atòmiques i ha hagut conflictes en el seu accés. Una vegada s'ha detectat el problema l'hem resolt mitjançant l'ús de semàfors.

4.4. Paral·lelitzar la implementació amb PVM

PVM és l'acrònim de Parallel Virtual Machine i consisteix en un conjunt d'eines i llibreries de pas de missatges, que ens serveixen per a desenvolupar programes en paral·lel. [13]

El sistema del PVM està basat en dues parts, la primera és el daemon pvmd que es troba en totes les màquines que formen la nostra màquina virtual. La segona és la biblioteca de PVM, que conté les primitives necessàries per tal que les tasques cooperin entre elles.

En aquest apartat ens centrarem en les rutines de PVM utilitzades per paral·lelitzar la implementació.

Control de processos

PVM permet que una tasca en llanci d'altres. De fet, l'escenari típic de PVM és que l'usuari engega una tasca que llançarà les demés sobre la màquina virtual, que a més és el cas d'implementació d'aquest model.

El node Mestre, engega tants processos Esclau com nodes té disponibles. Per a fer-ho hem utilitzat la comanda *pvm_spawn()*, concretant el node on volem que s'executi el procés, per tal d'evitar alguns problemes de rendiment que causa la ubicació automàtica.

El model de missatges

El model de comunicació de PVM, assumeix que qualsevol tasca pot enviar un missatge a qualsevol altra sense que hi hagi cap limitació en la mida o el nombre de missatges. Cal dir que es tracta d'una comunicació asíncrona que preserva l'ordre d'arribada dels missatges.

Per enviar un missatge amb PVM cal executar tres passos. En primer lloc, cal tenir un buffer de la biblioteca de PVM inicialitzat, cosa que es pot aconseguir mitjançant els mètodes *pvm_initsend()* o *pvm_mkbuf()*. El segon pas és empaquetar el missatge al buffer, mitjançant la comanda *pvm_pktipus()* on tipus denota el tipus de dades. El darrer pas és l'enviament del missatge mitjançant les comandes *pvm_send()* o *pvm_mcast()*. Pel que fa a la rebuda del missatge, en primer lloc s'utilitza la crida *pvm_recv()* i després els mètodes *pvm_upktipus()* necessaris. Cal dir que a l'hora d'enviar o rebre un missatge, no cal empaquetar tota la informació de cop, si no que es poden fer un seguit de crides *pvm_pktipus()* o *pvm_upktipus()*.

Com que el model de dades del que disposàvem era força complex, atès que totes les estructures contenien diversos tipus de dades, hem decidit enviar tota la informació com a caràcters, cosa que

ens permet fer un enviament i un tractament de les dades més ràpid però que en contrapartida ens limita el clúster a tenir una arquitectura homogènia.

A les versions anteriors del motor d'especulació, la mida dels missatges quedava limitada per una estructura estàtica que s'utilitzava com a missatge. En el nostre cas, necessitàvem poder enviar missatges de longitud variable, tot seguit mostrem algunes funcions que hem utilitzat per enviar els missatges en format de caràcters, i permetent que aquests tinguin qualsevol mida:

```
int pvm_escriuchar(int mida,char* dades){
      if (pvm_pkbyte(dades, mida,1) < 0) {
            printf("Problemes en l'enviament de caracters \n");
            pvm_exit();
            return -1;
         }
}
int pvm_llegeixchar(int mida,char* desti){
      if (pvm_upkbyte(desti, mida,1) < 0) {
            printf("Problemes en manipular el buffer al rebre caracters\n");
            pvm_exit();
            return -1;
      }
}
int pvm_envia_a_mestre(){
      if(pvm_send(pvm_parent(),1)<0){
            printf("Problemes al enviar missatge al pare %i\n",pvm_parent());
            pvm_exit();
            return -1;
      }
}
```

Figura 35: Funcions d'enviament de missatges

```
Enviar un missatge:

pvm_escriuchar(sizeof(int),(char *)&Nombre_Poligons);
pvm_escriuchar(sizeof(struct poligon)*Nombre_Poligons,(char*)poligons);
pvm_escriuchar(sizeof(int),(char *)&Nombre_regions);
pvm_escriuchar(sizeof(struct regio)*Nombre_regions,(char*)regions);
pvm_envia_a_mestre();

Rebre un missatge:

pvm_recv(-1,1);
pvm_llegeixchar(sizeof(int),(char *)&Nombre_Poligons);
tempPol=(struct poligon *)calloc(Nombre_Poligons,sizeof(struct poligon));
if(tempPol==NULL){ write(2,"ERROR TEMPOL\n",13); exit(-1);}
pvm_llegeixchar(sizeof(struct poligon)*Nombre_Poligons,(char *)tempPol);
... Tractament ...
free(tempPol);
pvm_llegeixchar(sizeof(int),(char *)&Nombre_regions);
        tempreg=(struct regio*)calloc(Nombre_regions,sizeof(struct regio));
if(tempreg==NULL){ write(2,"ERROR TEMPREG\n",13); exit(-1);}
pvm_llegeixchar(sizeof(struct regio)*Nombre_regions,(char*)tempreg);
...Tractament...
free(tempreg);
```

Figura 36: Exemple d'ús de les funcions d'enviament.

4.5. Generar els blocs pel Model Mestre-Mestre/Esclau-Esclau

En el cas que algun d'aquests blocs es pugui paral·lelitzar més des del seu interior li podem encarregar aquesta tasca a un node Mestre/Esclau. Aquest és el cas dels blocs del Preprocés i del First Shot. Tot seguit tractarem el graf de dependències d'aquests blocs.

En aquests blocs, en primer lloc cal determinar quin és el nombre de línies que a produir en cada cas i a continuació tractar-les (Figures).

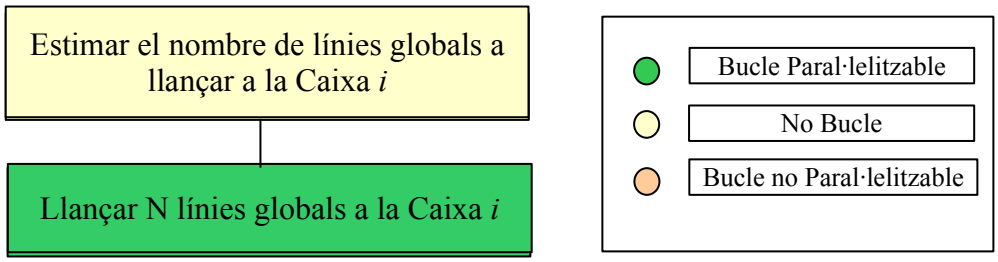


Figura 37: Blocs que s'executaran per cada subescena complexa

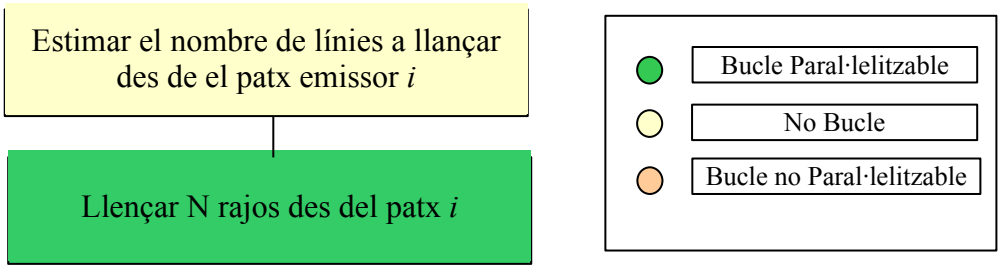


Figura 38: Blocs que s'executaran per cada patx emissor

Com que dividir el bloc dels nodes esclaus per cada raig, pot comportar un cost excessiu en la transmissió de dades, hem fet que cada node esclau tracti diverses línies.

4.6. Implementació amb el model Mestre-Mestre/Esclau-Esclau amb PVM

En aquesta fase, hem afegit el codi del node Mestre/Esclau, que només estava implementat per a l'ús concurrent i l'hem modificat per executar-lo de manera paral·lela amb PVM. També hem

generat el graf de dependències del programa Mestre/Esclau tal i com es descriu en l'apartat anterior.

A l'hora d'incloure els nodes del tipus Mestre/Esclau, hem decidit doblar el mètode per iniciar els blocs, que ens permet d'una banda un codi més senzill i entenedor, que prioritzi l'execució dels nodes Mestre/Esclau i que redueixi el temps de gestió per iniciar un procés.

A més a més, hem utilitzat els grups de processos de PVM per tal de crear la jerarquia de nodes, d'aquesta manera hem pogut fixar per a cada Mestre/Esclau on s'han d'executar els seus nodes Esclaus.

També cal dir, que en la implementació d'aquest model, els fitxers amb les línies globals generades al Preprocés són escrits a disc pels nodes Mestre/Esclau, mentre que en el model anterior eren els Esclaus els encarregats d'escriure al disc dur.

4.7. Optimització del codi

En aquest apartat parlarem de la optimització que hem fet a la implementació, amb l'objectiu d'aconseguir un temps de resposta menor. Concretament hem millorat el grau de paral·lelisme del bloc de Multipath i els accessos al disc dur.

4.7.1. Millora sobre el grau de paral·lelisme

Si ens fixem detingudament en el mètode Multipath, podem observar que només es poden executar alhora les caixes del mateix nivell. Ja que a les regions de les caixes dels nivells posteriors sel's hi modifica l'energia entrant de les regions angulars. A més a més, a la següent execució de les caixes pares s'utilitza l'energia sortint de les regions virtuals de les caixes filles, de manera que la següent iteració de les subescenes pares no es pot executar fins que han finalitzat les subescenes filles. De manera, que hi ha dependència de dades entre els diferents nivells de la jerarquia de subescenes i les diferents iteracions.

Si les modificacions que es fan a les regions angulars no afecten fins a la següent iteració, com és el cas de la potència de sortida, totes les caixes es podrien tractar alhora.

Per aquest motiu hem decidit duplicar a les regions les variables de potència de sortida i d'entrada.

Així tenim les potències que s'han acumulat a les regions la iteració anterior per utilitzar-les en l'actual, i podem produir el valor de les potències d'aquesta iteració sense interferències, per utilitzar-les en la següent iteració. Tot i que continua havent-hi dependència de dades entre les potències d'una iteració i la següent, no n'hi ha entre el tractament de les caixes, cosa que ens permet augmentar el grau de paral·lelisme.

A més, ens ha semblat que podia ser interessant que diversos processadors tractessin una part de les línies, tal i com hem fet en els blocs del Preprocés i del First Shot. Si analitzem l'algorisme, però, podem veure que el tractament que es fa de l'energia unshot ens genera dependències entre el tractament de les diferents línies. Per resoldre-ho, hem afegit un nou atribut als patxos reals de manera que en una iteració es distribueix la energia unshot que s'ha rebut en la iteració anterior repartida entre totes les línies, i en l'altre atribut es va acumulant l'energia unshot d'aquesta iteració per distribuir-la a la propera. En aquest punt, però, el que ens limita és el motor d'especulació que no permet executar els blocs de tercer nivell de manera paral·lela.

Un dels problemes respecte a la implementació original, és que la potència entrant a les darreres iteracions no arribarà als polígons, cosa que hem resolt afegint tantes iteracions com nivells de jerarquia.

4.7.1.1 Loop-unroling del Multipath

Una de les limitacions del motor d'especulació, és que només pot tractar el paral·lelisme de dos bucles niats, ja que els nodes Mestre/Esclau no poden usar altres nodes Mestre/Esclau. Anteriorment, hem explicat com tractar les línies de les subescenes de manera paral·lela, i hem comentat que no era aplicable per aquesta limitació.

Tot i això, hem observat que en no poder paral·lelitzar el primer bucle de l'algorisme Multipath, (ja que una iteració depèn de la anterior), podem aplicar un loop-unroling de manera que el bloc que rep el node Mestre/Esclau és el tractament d'una caixa en l'algorisme del Multipath. Així hem eliminat la limitació del motor d'especulació i hem aconseguit un major grau de paral·lelisme.

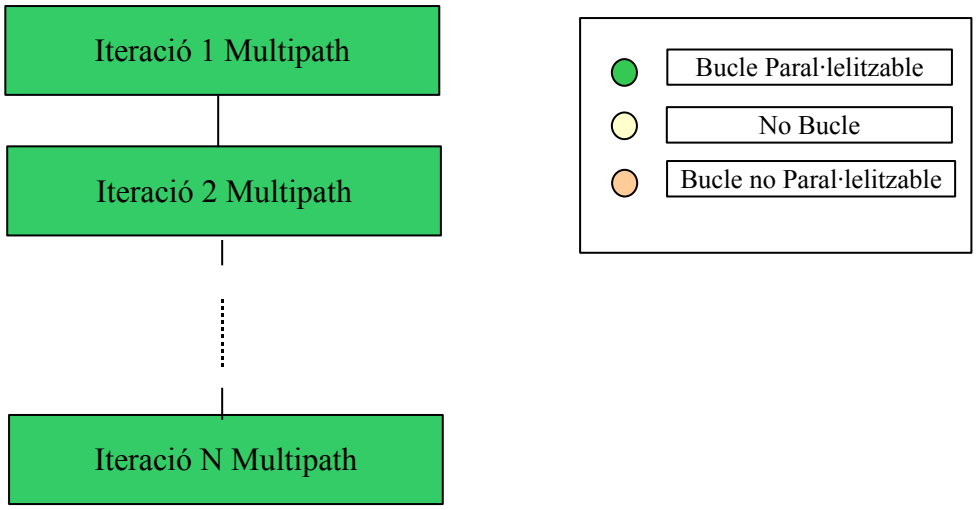


Figura 39: Loop-unrolling del Multipath

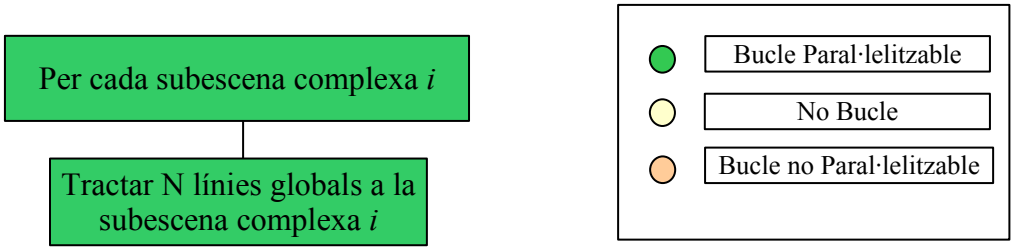


Figura 40: Cada iteració conté els següents blocs

En aquesta ocasió, tornem a obtenir el paral·lelisme mitjançant l'especulació de les variables d'inducció dels bucles.

4.7.2. Accesos al disc dur

La implementació de l'algorisme utilitza el disc dur per emmagatzemar les línies globals generades. A l'original, s'accedia al disc dur cada vegada que es tractava una línia. A [21] trobem uns experiments realitzats sobre l'accés a discs durs remots i locals amb diferents mides de buffer.

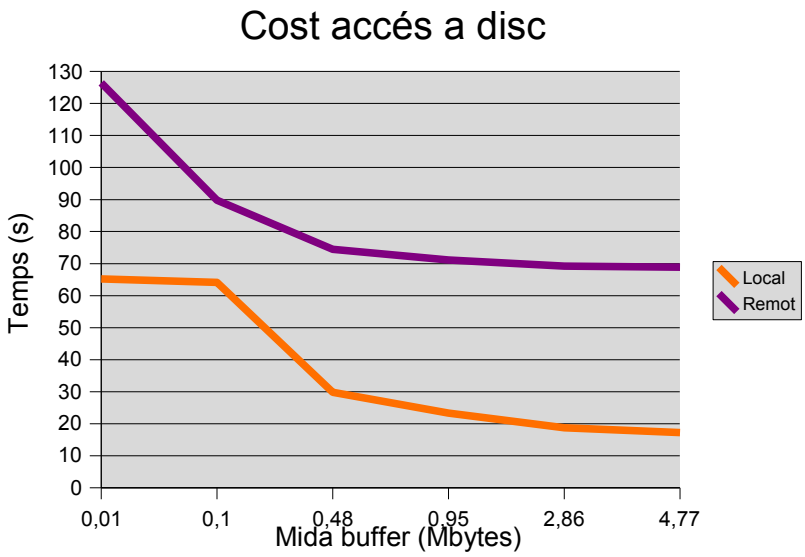


Figura 41: Cost d'accés a disc en funció de la mida del buffer a l'hora de copiar un fitxer, en aquest cas de 332 Mbytes

Podem observar que accedir a un fitxer amb un buffer massa petit suposa una pèrdua de rendiment en la aplicació, per aquest motiu en lloc d'escriure les dades a disc directament des del moment que es generen, les hem emmagatzemat en un buffer des del qual s'escriu al disc un cop està ple o finalitza el contingut a emmagatzemar. Pel que fa a la lectura, hem procedit a llegir un bloc de dades, enlloc de només una línia global. D'aquesta manera hem reduït el nombre d'accessos a disc i hem millorat el rendiment de l'aplicació de manera considerable.

4.8. El simulador

Inicialment disposàvem d'un simulador que usava mitjanes pel temps d'execució de cada bloc, i que no feia ús dels nodes Mestre/Esclau, en veure que era complicat fer les modificacions i que la millora del simulador quedava fora dels objectius d'aquest projecte vam decidir fer-ne un de nou però més senzill.

Cal dir, que com l'algorisme a paralel·litzar és de gràfics, la majoria de dades no eren enteres ni especulables, a excepció de les variables d'inducció dels bucles for en què no es falla mai la predicció (sempre i quan l'increment i el límit de la condició siguin constants), ja que coneixem l'increment i la condició d'acabament. Per aquest motiu, no necessitem que el simulador tracti els errors de predicció.

Per a cada bloc hem tingut en compte els següents costos:

El temps de gestió d'iniciar el bloc (M) , el d'enviar el missatge (X), l'execució del bloc (E o M/E), el d'enviar el resultat (X) i finalment la escriptura al node Mestre (M). A més, el cost d'enviar el missatge l'hem separat en dues parts, el delay i el cost de transmissió.

El simulador consta principalment de dues taules dinàmiques, on es van emmagatzemant els processos llançats, una és pels processos esclaus i la altre pels processos Mestre/Esclau. Els processos es poden llançar en les mateixes condicions que en el motor d'especulació, quan es tenen les dades que necessita el bloc (ja s'hagin obtingut mitjançant especulació, o per que les instruccions productores han acabat) i tenim un node que el pugui executar. A la taula s'hi emmagatzema el bloc amb l'hora de finalització, que inclou el temps de xarxa i el de CPU. A mesura que es llancen processos incrementa el temps actual del simulador en funció del cost de la gestió inicial del procés.

Quan el simulador no pot llançar més procesos procedeix a rebre els missatges. Si no hi ha cap missatge disponible el temps actual del simulador s'igualarà amb el del primer missatge a rebre, i se l'incrementarà pel cost de gestió al finalitzar el procés. Si hi havia algun missatge disponible, només caldrà incrementar el cost de gestió. Tot seguit el simulador mira si pot llançar un altre procés.

També s'ha inclòs com a funcionalitat la simulació dels costs de xarxa i de CPU en funció dels anys seguint les lleis de Glider i Moore.

$$CPU_{ANYS} = \frac{CPU_0}{2^{\frac{ANYS}{1.5}}} \quad T_{ANYS} = \frac{T_0}{3^{ANYS}}$$

Equacions 15 i 16: Variació del temps de CPU i de transmissió de dades en funció dels anys.

Hem provat el simulador amb les dades temporals produïdes per execucions amb un node de cada, i per la simulació de la mateixa estructura ens donava un error del 5%. Però a mesura que s'incrementaven els nodes, el simulador augmentava l'error i anava més depressa que l'execució real. Hem suposat que principalment es deu al cost d'accés al disc remot, que augmenta si hi

accedeixen més nodes alhora, i es podria resoldre disposant d'un disc local a cada node, tot i això necessitàvem comprovar que el simulador funcionava correctament, i per aquest motiu hem fet la traça d'una execució d'exemple per comprovar-ho, la qual mostrem tot seguit.

4.8.1. Traça d'exemple

Disposem d'una aplicació dividida en 9 blocs dels quals de l'1 al 7 són blocs Esclau que no tenen cap dependència entre ells, i els blocs 8 i 9 són blocs Mestre/Esclau que depenen dels blocs de l'1 al 7, i a més a més les dependències dels blocs 8 i 9 no són especulables. Els blocs 8 i 9 estan subdividits en dos, al primer (bloc 8' o 9') només se'n fa una iteració i al segon (28 o 29) se'n fan 5 que no tenen dependències entre ells i depenen del bloc anterior (8' o 9').

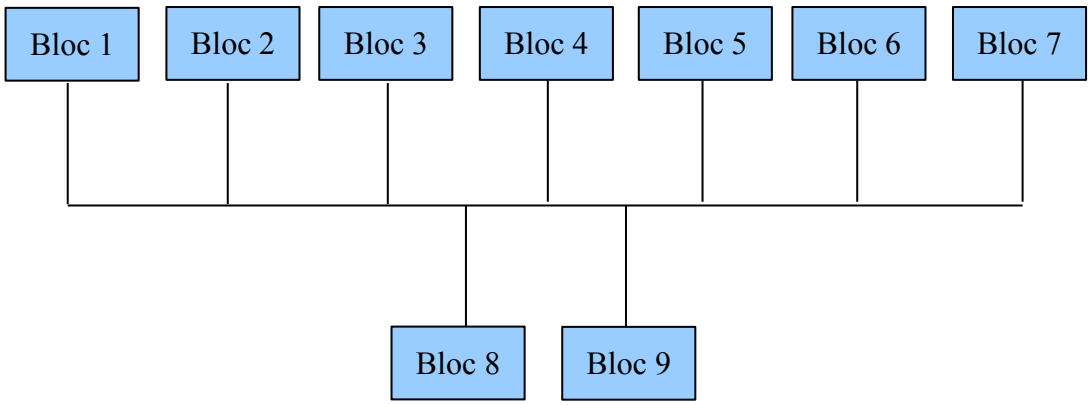


Figura 42: Graf de dependències entre blocs pel node Mestre

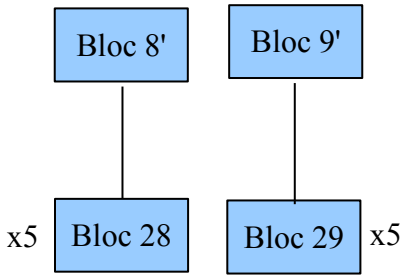


Figura 43: Graf de dependències entre blocs pel node Mestre-Esclau

Pel que fa al nombre de nodes, disposem d'un node Mestre amb 3 nodes Esclau i 2 de Mestre/Esclau. Cada node Mestre/Esclau té 5 nodes Esclau. El temps de gestió dels processos és d'un segon en tots els casos, i els d'execució són de 5 segons pels blocs de l'1 al 7, d'un pels blocs 8' i 9', i de 10 segons pels nodes 28 i 29. Per tal que l'exemple sigui més entenedor hem establert com a cost de transmissió 0 segons.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Bloc 1	M	E_1	E_1	E_1	E_1	E_1	M													
Bloc 2		M	E_2	E_2	E_2	E_2	E_2		M											
Bloc 3			M	E_3	E_3	E_3	E_3	E_3			M									
Bloc 4								M	E_1	E_1	E_1	E_1	E_1	M						
Bloc 5										M	E_2	E_2	E_2	E_2	E_2	M				
Bloc 6												M	E_3	E_3	E_3	E_3	E_3	M		
Bloc 7															M	E_1	E_1	E_1	E_1	E_1
Bloc 8																				
Bloc 9																				

Taula 3: Execució des del node Mestre (part 1)

21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
M																						
	M	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	ME_1	M	
		M	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	ME_2	M

Taula 4: Execució des del node Mestre (part 2)

Bloc 8	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Bloc 8	M	E_1	M																
$Bloc\ 28_1$				M	E_1	E_1	E_1	E_1	E_1	E_1	E_1	E_1	E_1	E_1	M				
$Bloc\ 28_2$					M	E_2	E_2	E_2	E_2	E_2	E_2	E_2	E_2	E_2	E_2	M			
$Bloc\ 28_3$						M	E_3	E_3	E_3	E_3	E_3	E_3	E_3	E_3	E_3	E_3	M		
$Bloc\ 28_4$							M	E_4	E_4	E_4	E_4	E_4	E_4	E_4	E_4	E_4	E_4	M	
$Bloc\ 28_5$								M	E_5	E_5	E_5	E_5	E_5	E_5	E_5	E_5	E_5	E_5	M

Taula 5: Execució del bloc 8 en el node Mestre-Esclau, el bloc 9 és equivalent.

Com podem veure a les taules, el funcionament és molt semblant al de l'arquitectura segmentada. Per conèixer el temps que triga els blocs Mestre/Esclau cal resoldre'ls apart, en aquest exemple podem comprovar que triguen 19 segons, i que l'execució de l'aplicació en dura 43, igual que al simulador.

4.9. Obtenir resultats

La darrera part ha consistit en usar la implementació paral·lela de l'algorisme de gràfics per tal d'estudiar els costos temporals dels diferents blocs i de les seves etapes. Cal dir que en el clúster

sobre el que hem realitzat les proves finals no podíem usar gaire quota de disc dur[7] de manera que el nombre de línies globals quedava acotat i no hem mesurat execucions realment complexes, encara que a mesura que augmenten el nombre de línies globals i locals el grau de paral·lelisme de l'algorisme augmenta, ja que no hi ha cap mena de dependència de dades entre elles.

Per a cada execució de cada bloc hem mesurat el temps de gestió inicial, el temps de gestió final, el temps d'execució en el node esclau i el nombre de bytes enviats.

Hem configurat els blocs del First Shot i de la generació de línies globals perquè llancin un màxim d' 1, 0.5, 0.25 i 0.1 milions de rajos i poder estudiar la diferent mida d'aquests blocs.
Cada configuració diferent la hem executat 10 vegades i n'hem calculat l'interval de confiança.
Pel que fa al cost de la xarxa, hem avaluat el temps que triguen els missatges de PVM per tal d'establir el temps de comunicació, on també hem intentat mesurar-ne el delay.

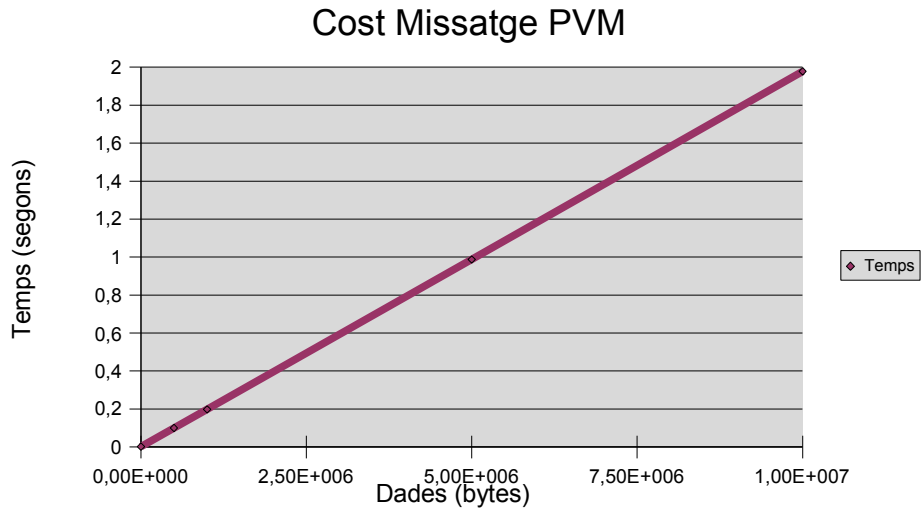


Figura 44: Cost dels missatges amb PVM

D'on hem obtingut que el cost d'enviar 10^3 bytes és de 0.2 ms, i el delay l'hem estimat a 2 ms.

Els experiments que després hem realitzat amb el simulador partint de les sortides obtingudes han sigut: estimar el cost de resposta quan augmentem el nombre de nodes del clúster, simular la situació dels pròxims anys si es compleixen les lleis de Moore i de Glider, i calcular el grau de paral·lelisme i el cost per raig de les diferents configuracions pels blocs de generació de les línies globals, del First Shot i del Multipath. En el següent capítol comentarem els resultats obtinguts en cadascun d'ells.

[7] Es tracta d'una compte d'estudiant a BAS

5. Resultats

En aquest capítol, mitjançant l'ajuda del simulador, estimarem quin seria el temps de resposta de l'algorisme en funció del nombre de nodes, i també observarem quina serà la tendència en els propers 10 anys segons les lleis de Glider i de Moore.

En primer lloc, cal dir que el grau de paral·lelisme que es pot obtenir amb aquest algorisme depèn de molts factors. El primer és el nombre de subescenes que conté l'escena, el segon el nombre de patxos reals emissors de llum, que dependrà del grau de discretització indicat per l'usuari i en tercer lloc depèn del nombre de línies de cada tipus que es generin.

En el nostre cas d'estudi hem utilitzat una escena que consta de 4 subescenes més la principal, la font de llum ha quedat discretitzada en 16 patxos virtuals. La execució no és gaire llarga, ja que hem hagut de limitar el nombre de línies globals a causa del poc espai de disc del que disposàvem, tot i això, sabem que per una execució més llarga el grau de paral·lelisme serà igual o major.

5.1. Temps de resposta en funció dels nodes

Pel que fa al nombre de nodes de cada tipus que s'utilitzen, hem decidit anar-los incrementant de manera equitativa, de manera que en cada mesura s'incrementen el nombre de nodes Esclaus per cada Mestre o Mestre/Esclau, i el nombre de nodes Mestre/Esclau. Hem pres la decisió perquè el temps de resposta que s'obté depèn de la configuració del clúster, i d'aquesta manera es pot observar la tendència, encara que en funció de la divisió en blocs la configuració es podria ajustar molt més per tal d'obtenir un millor rendiment.

Per saber el nombre de nodes que utilitzem en cada cas podem usar la següent equació, on tenim 1 node Mestre, N nodes Mestre/Esclau, N nodes Esclau per cada Mestre esclau (N^2 nodes Esclau entre tots els nodes Mestre/Esclau) i N nodes Esclau pel node Mestre:

$$Nodes=1+N+N+N*N=N^2+2N+1=(N+1)^2$$

Equació 18: Nombre de nodes en funció d'N.

En les següents figures enlloc d'indicar el nombre de nodes, indicarem la N.

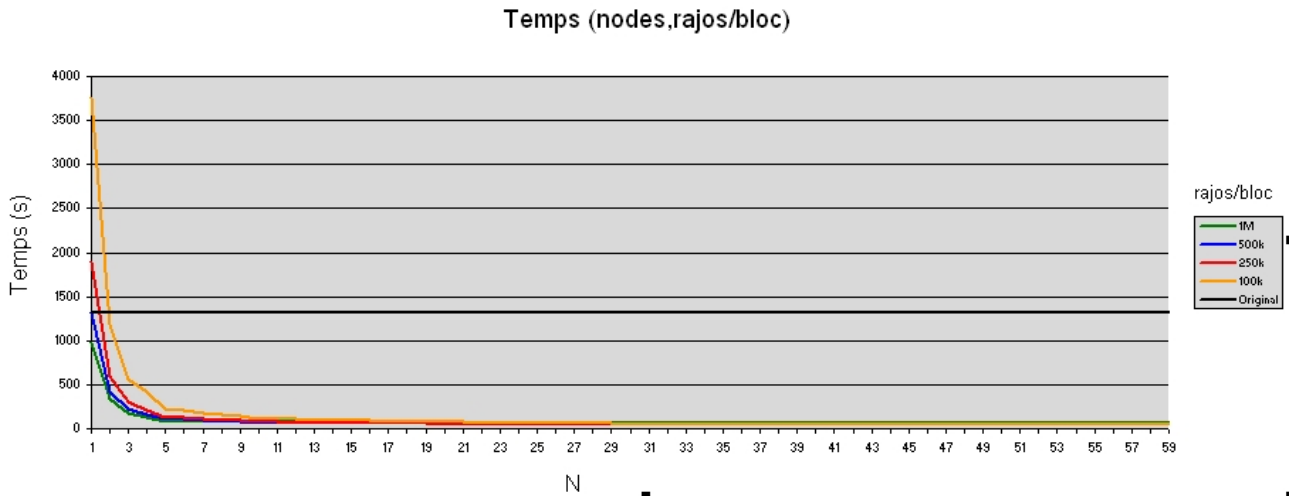


Figura 45: Temps d'execució en funció de N i de les línies per bloc

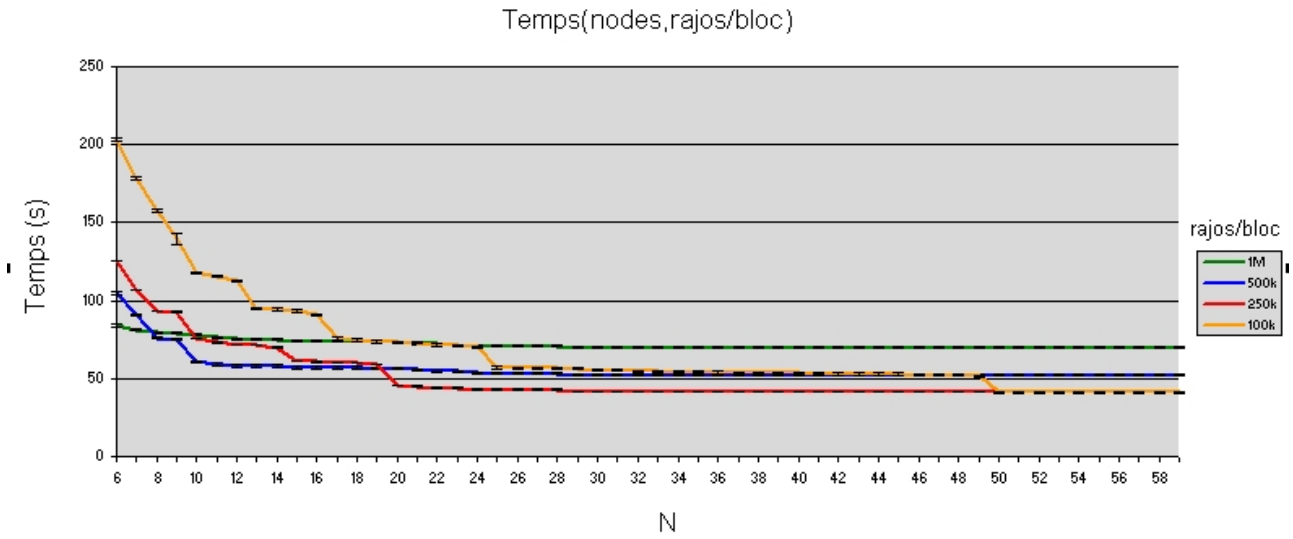


Figura 46: Temps d'execució en funció de N i de les línies per bloc. Gràfic ampliat.

D'entrada podem observar que en l'execució de N=1, que tot i estar distribuida no és paral·lela (ja que els diferents nodes no treballen de manera simultània), obté millors resultats que l'original quan la subdivisió dels blocs ha sigut d'un milió de línies, tot i els costos de comunicació. Això es deu a l'optimització de l'accés a disc.

En funció de la mida dels blocs, podem observar que com més petit és el bloc convergeix en un temps menor a mesura que augmentem els nodes, però es necessiten més nodes per arribar-hi. Per exemple, a partir de N=5 l'execució amb els blocs d'un milió de línies millora molt poc en augmentar el nombre de nodes. Mentre que l'execució per 0'1 milions de línies convergeix cap a N=50.

En el gràfic ampliat, també podem veure que la millora del temps de resposta es fa de manera

esglaonada en augmentar N. Això es deu a que el temps de resposta es redueix significativament quan el nombre de blocs paral·lelitzables és divisible pel nombre de nodes que els poden tractar.

Com que en els gràfics anterior (Figura 45 i 46), només mostravem l'execució d'un tipus de configuració, tot seguit mostrem el resultat de diferents configuracions sobre 26 nodes (Figura 47). En aquest cas, podem observar que en general la millor configuració és la de 1 Mestre, 4 Mestre/Esclau, 4 Esclaus per a cada Mestre/Esclau i 5 Esclaus pel node Mestre, que a més, és la més semblant a la dels gràfics anteriors, encara que si agaféssim més nodes, podríem veure que per l'execució tractada, és millor augmentar els Esclaus enlloc dels Mestre/Esclau (ja que el nombre màxim de blocs paral·lels destinats als nodes Mestre/Esclau és de l'orde d'uns 20).

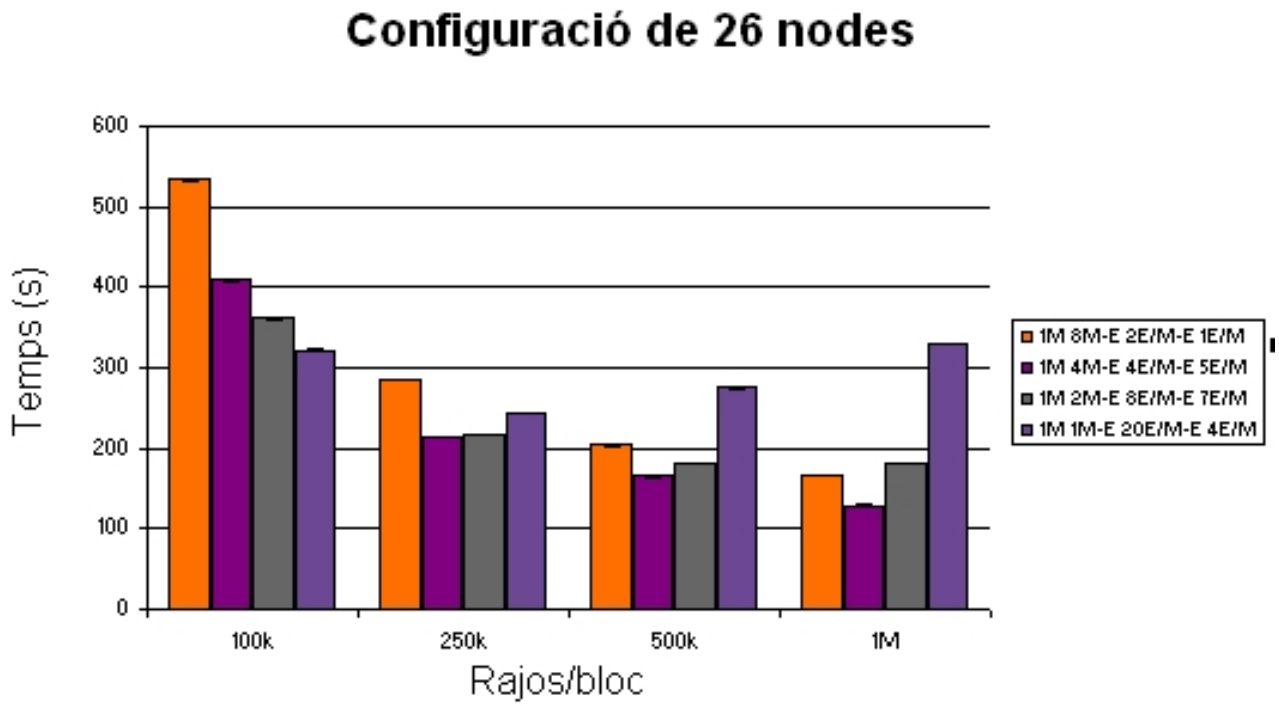


Figura 47: Temps per diferents configuracions per 26 nodes

5.2. Llei de Moore i de Glider

En les següents gràfiques mostrem el temps de resposta en els pròxims anys en el supòsit que les lleis de Moore i de Glider segueixen complint-se.

En primer lloc mostrem l'evolució del temps d'execució en un entorn amb N=10 , i tot seguit amb un on no hi ha limitació amb el nombre de nodes a utilitzar. Com que el resultat per les diferents mides dels blocs al generar les línies no és apreciable en les primeres gràfiques, també les afegim

amb un eix logarítmic que permet veure amb més claredat la tendència.

Les gràfiques ens permeten observar que la tendència, és que l'augment del cost de transmissió que afegeix la divisió en blocs més petits no afectarà en el futur.

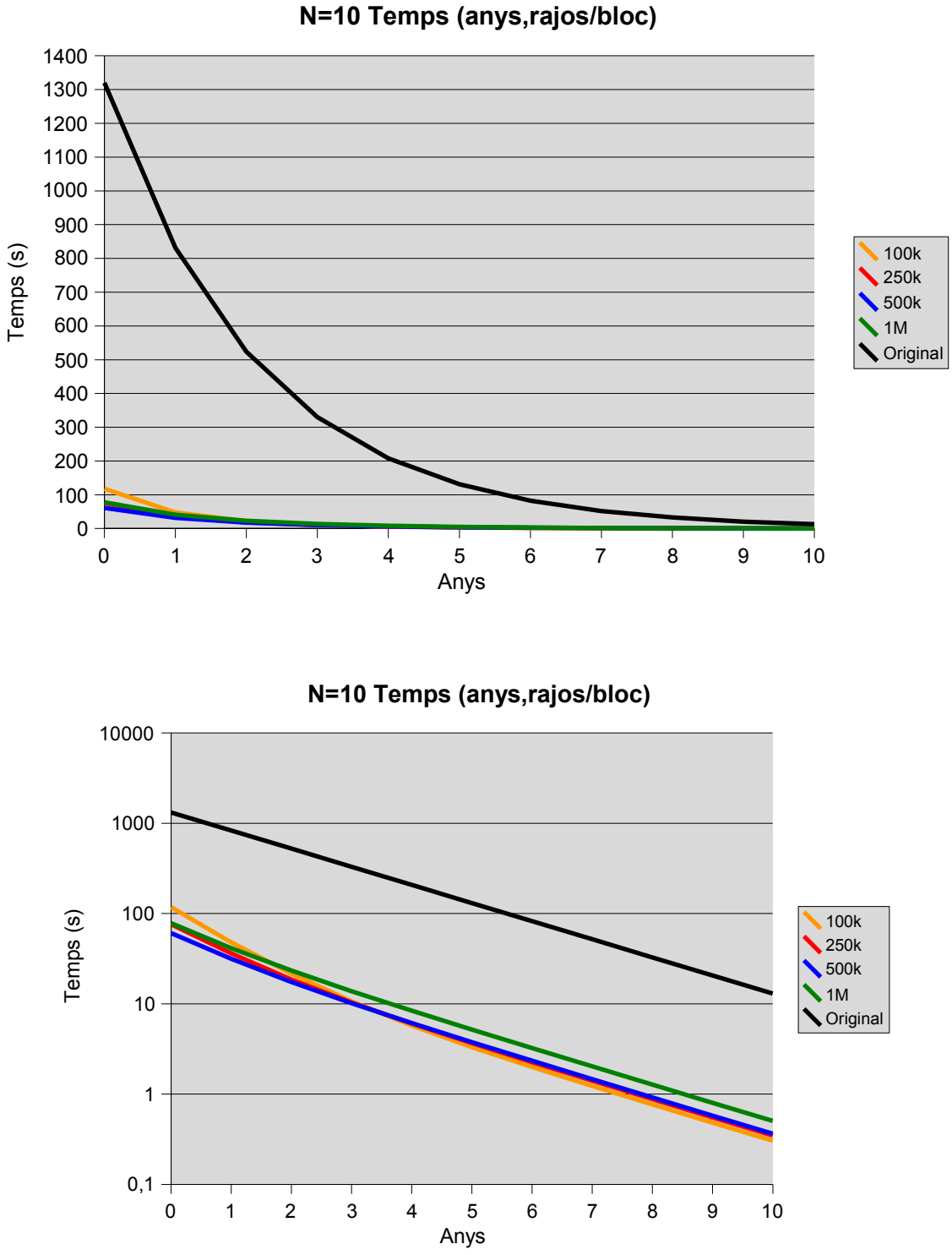


Figura 48: Cost temporal en funció dels anys per un total de 121 nodes

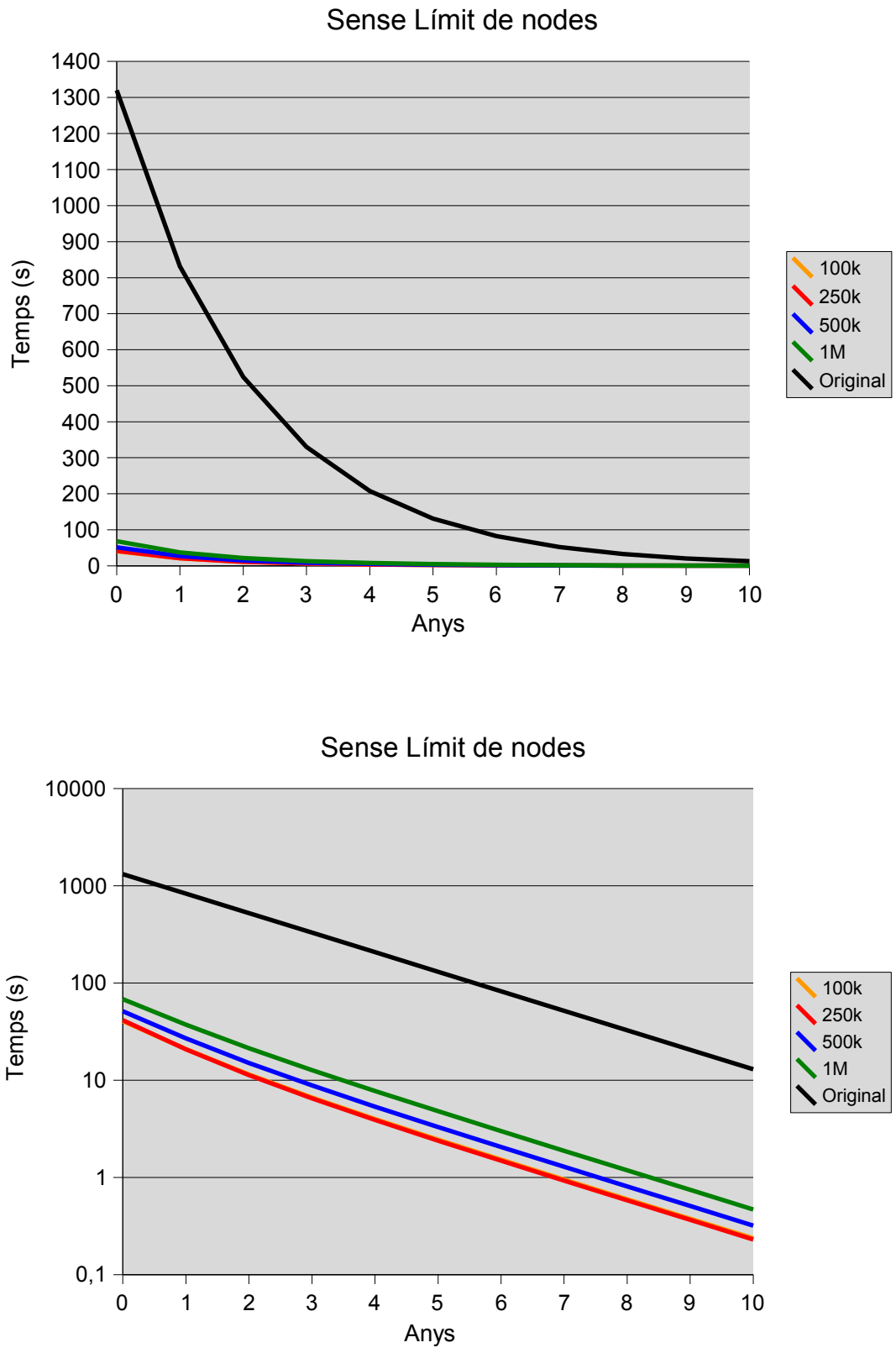


Figura 49: Cost temporal en funció dels anys sense límit de nodes

En la figura 48, podem observar que tot i que actualment és més costosa l'execució amb els blocs més petits, a la llarga, en reduir-se més el temps de transport que el d'exeució la situació canviarà, ja que els blocs més petits ofereixen més grau de paral·lelisme. També podem observar que aquesta

millora tè un límit, doncs no hi ha gaire diferència entre l'execució dels blocs de mida de 0'1 milions de línies i els de 0'25 milions, un dels motius, és que el delay no l'hem modificat en funció dels anys, ja que la llei de Glider ens parla de l'augment de l'ample de banda, i que distingir els diferents temps que formen el delay seria molt complicat.

5.3 Temps dels blocs de Multipath i Preprocés

Com hem vist en el model matemàtic, a mesura que el nombre de blocs a executar en paral·lel creix, el seu temps mig tendeix al de gestió, per aquest motiu ens ha semblat adequat estudiar els diferents temps de bloc de les tasques que hem dividit en diverses mides: el preprocés, el first shot, i el multipath. No hem pogut estimar els costos temporals pel first shot, atès que el nombre de rajos no era suficientment elevat per tal de poder apreciar tots els casos, encara que per la naturalesa del bloc, sabem que el seu cost de gestió no depèn del nombre de línies llançats, i tampoc el cost de transmissió, que són constants per totes les mides. Pel que fa al cost de l'execució del bloc, hauria de tenir un comportament similar al preprocés, per aquest motius, ens ha semblat que estudiant només aquests dos blocs, no era necessari estudiar-ne el tercer.

Pel que fa als temps de gestió, no sembla que hi hagi una diferència significativa en funció del nombre de línies a tractar. Segurament, en el cas del Preprocés, si que hi pot haver un increment significatiu a partir de certa mida en el segon temps de gestió quan l'augment de la mida del buffer no faci reduir el temps d'accés al disc dur (com hem vist a l'apartat 4.7.2 amb la Figura 41).

Hem pogut comprovar que el cost mig per raig en cada bloc és inversament proporcional a la mida del bloc, principalment perquè en tots els casos les dades d'entrada tenen la mateixa mida, cosa que fa que el temps de CPU i de xarxa que s'hi destina en total augmenti en funció del nombre de blocs. Del que en podem extreure que com més línies llancem en aquests blocs, major serà el seu grau de paral·lelisme, encara que disposarem de menys blocs per tractar en paral·lel. Si les lleis de Moore i de Glider segueixen la mateixa tendència, quan el cost de la xarxa no sigui significatiu, ens interessarà aplicar la mida del bloc més petit que ens permeti aprofitar al màxim el nombre de nodes que disposem per resoldre l'execució objectiu.

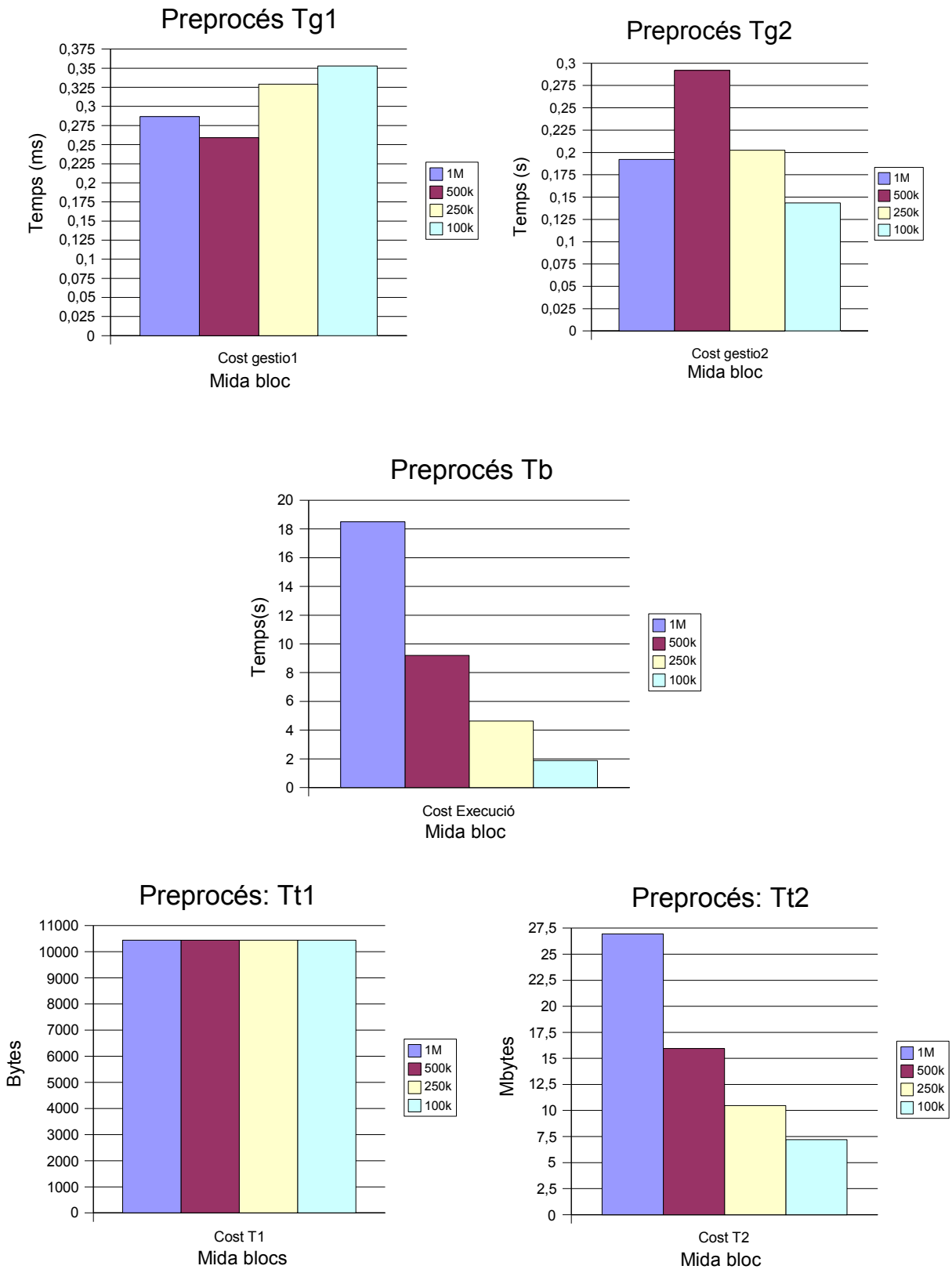


Figura 50: Cost temporal dels blocs de preprocés en funció de la mida del nombre de línies llençades per bloc

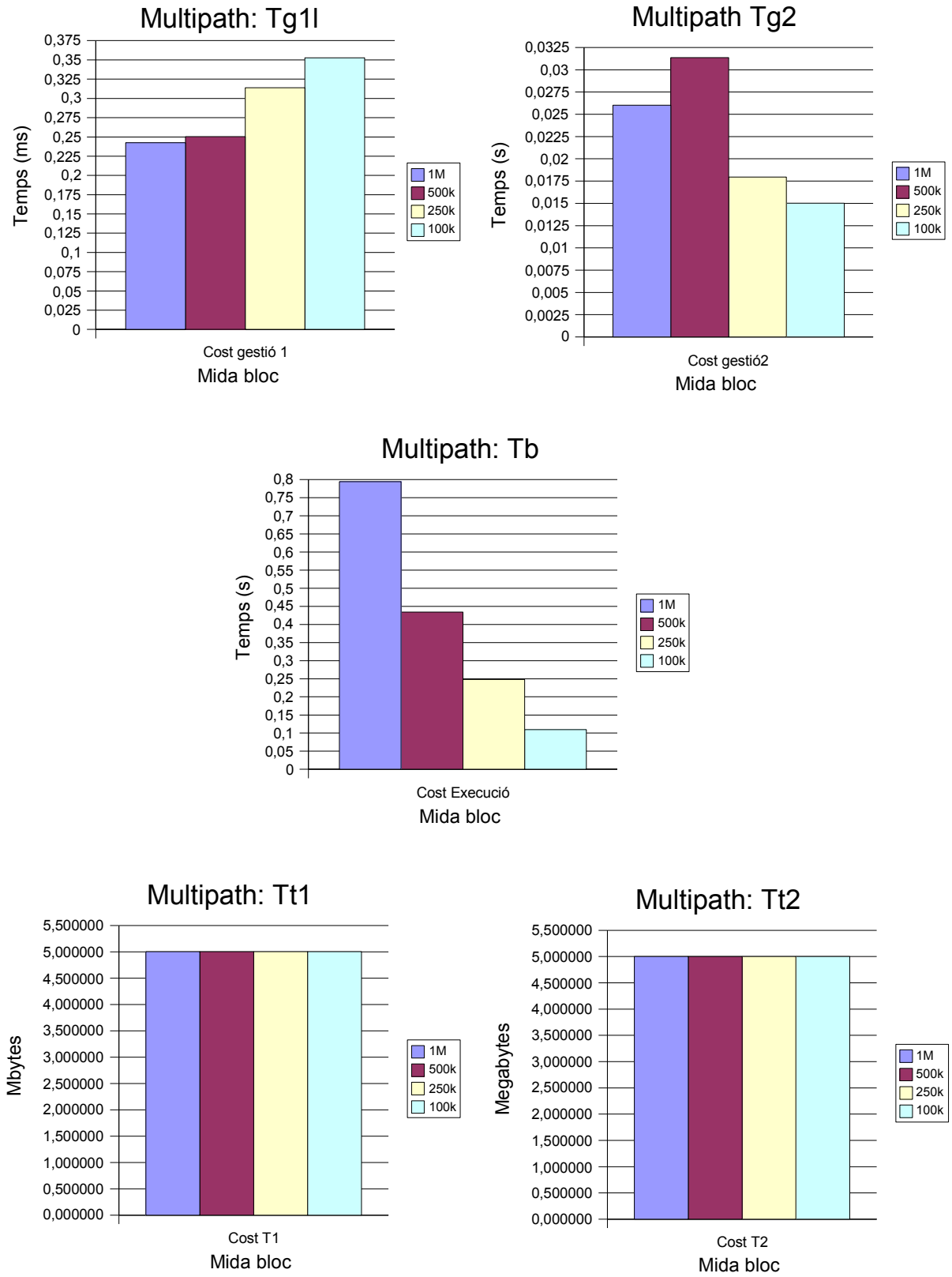
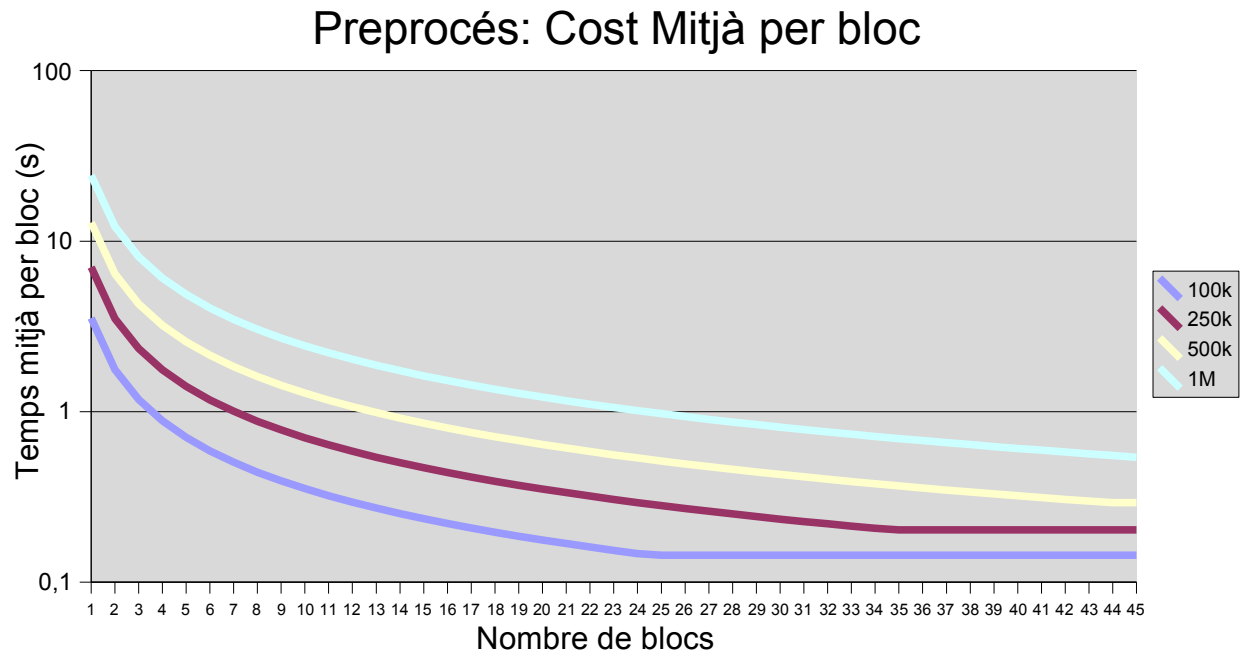


Figura 51: Cost temporal dels blocs de multipath en funció de la mida del nombre de línies llençades per bloc



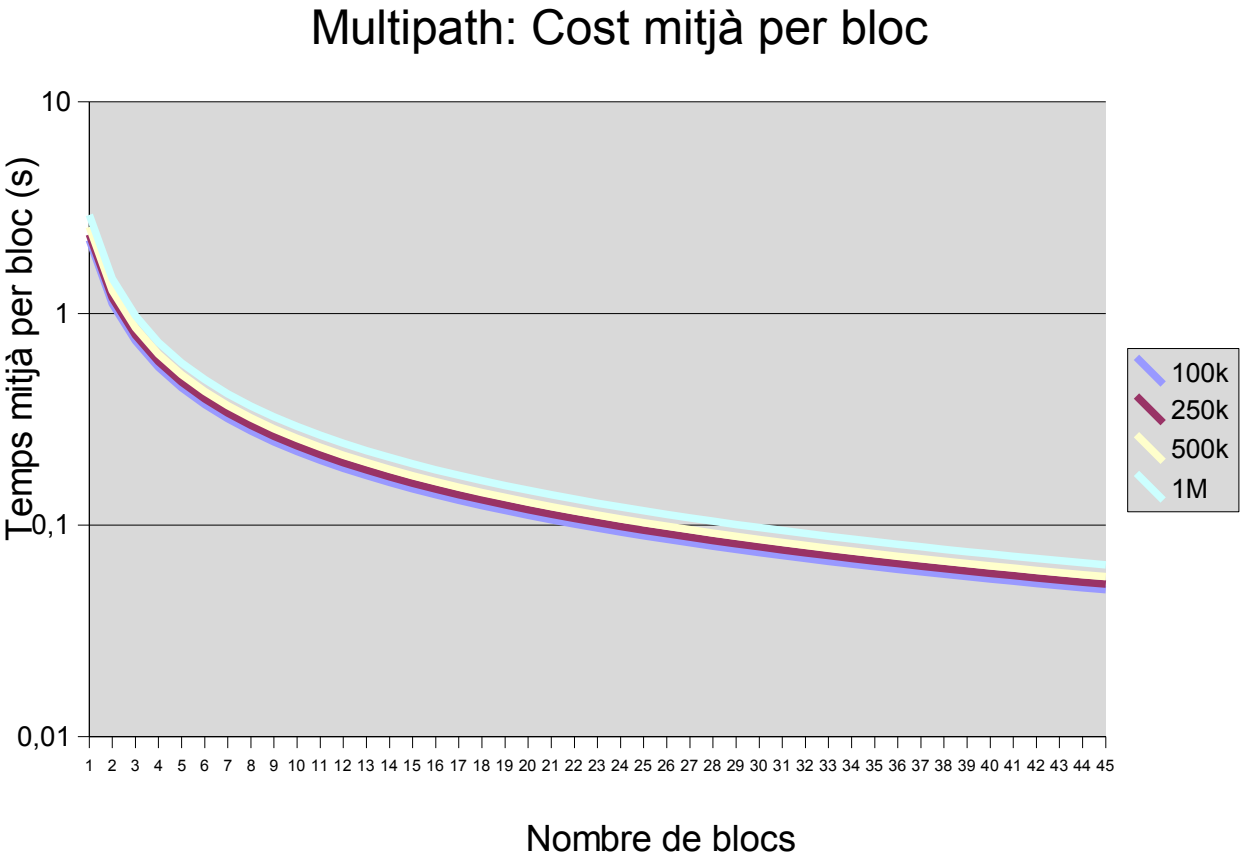


Figura 52: Cost Mitjà per bloc del Multipath i el Preprocés

6. Conclusions

L'objectiu principal d'aquest projecte era paral·lelitzar aplicant especulació a l'algorisme *Fast Multipath Radiosity Using Herarchical Subscenes,* cosa que hem aconseguit amb uns resultats millors als que esperàvem, ja que hem passat d'un temps d'uns 1550 segons a uns 42. Mitjançant la llei d'Amdhal podem dir que hem obtingut un grau de paral·lelització respecte a l'implementació original del 97%.

A més a més, hem utilitzat els resultats temporals obtinguts per tal d'obtenir conclusions i noves idees sobre el motor d'especulació, del que n'hem trobat algunes mancances que s'hauran de considerar si se'l volgués utilitzar en un entorn real. També hem tingut algunes idees per millorar tant el rendiment de les màquines que usin el motor d'especulació i l'execució en paral·lel de l'algorisme de gràfics (veure treball Futur).

Pel que fa als estudis sobre la mida del blocs, hem vist que es tracta d'una decisió complexa que dependrà de l'execució en particular que vulguem paral·lelitzar i dels recursos que tinguem disponibles.

Sobre la tendència en el temps dels pròxims anys, si es compleix el que anuncien les lleis de Moore i de Glider, podrien arribar-se a executar aplicacions com aquesta en temps real.

7. Treball futur

7.1. Estructura dinàmica del clúster

Com hem observat en els resultats, el fet de tenir una estructura del clúster fixa fa que en moltes ocasions els nodes Esclau no executin res mentre podrien estar contribuint en l'acceleració de l'execució, un exemple clar, és quan disposem d'un node de cada tipus en la paral·lelització d'aquest algorisme, on els node Esclau del Mestre i el del Mestre/Esclau no treballen mai alhora, de manera que l'execució amb estructura fixa és distribuida però no paral·lela.

Aquesta característica seria millorable, si la jerarquia dels nodes fos dinàmica, cosa que es podria resoldre fent que els esclaus estiguessin compartits o que els nodes Mestre i Mestre/Esclau es deixessin els nodes Esclaus. Aquesta idea també ens ha portat a pensar, que podríem mirar de donar prioritat als blocs en funció de la possibilitat d'encert amb les variables d'entrada, el temps d'execució del bloc i el nombre de blocs que al finalitzar aquest bloc es podran executar.

Aquest treball podria ser complex però permetria aconseguir un rendiment més elevat amb el mateix nombre de nodes.

Per il·lustrar-ho, mostrem quin seria el rendiment si apliquéssim aquest procediment en la implementació de l'algorisme en què disposem un node de cada i amb el supòsit que no incrementa el cost de gestió.

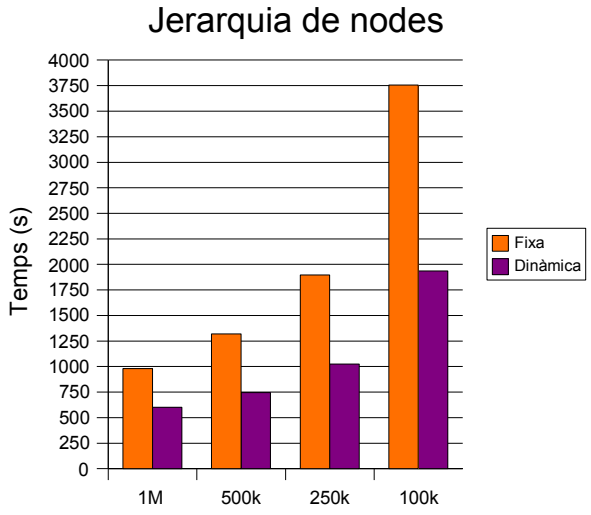


Figura 53: Temps amb estructura de nodes dinàmica i fixa

7.2. Aplicar el mètode del polígon associat mitjançant el patró strategy

Dins de l'algorisme *Fast Multipath Radiosity Using Hierarchical Subscenes*, hi ha un mètode per accelerar el First Shot, que consisteix en associar un polígon als rajos que intersequen una regió angular, però que a canvi el fa dependre del preprocés.

La idea principal, és que si totes les línies globals llançades que han intersecat la regió R tenen com a la següent intersecció el polígon P, llavors associem el polígon P a la regió R. D'aquesta manera quan un raig entri a una subescena per la regió R, és molt probable que la seva següent intersecció sigui el polígon P. Aquest còmput permet reduir aproximadament a la meitat el cost del first shot.

Usar-lo ens permetria reduir el temps del first shot, però a canvi generaria dependències de dades no especulables entre aquests dos blocs. En funció de l'escena i l'entorn d'execució, incloure'l podria augmentar el rendiment (si no tenim suficients nodes per executar els blocs del first shot i del preprocés alhora) o bé disminuir-lo (si disposéssim de suficients nodes) .

Per mirar d'obtenir un millor temps de resposta en qualsevol cas, es podria implementar l'algorisme del polígon associat a les regions angulars i utilitzar un patró *Strategy* per tal d'usar l'algorisme de First Shot més convenient en cada cas. D'aquesta manera no limitaríem el grau de paral·lelisme de la implementació i els blocs del First Shot que s'executin un cop ha finalitzat la generació de línies globals en podran teure profit, cosa que aprofitaria les avantatges dels dos enfocaments.

8. Bibliografia

[1] Apunts Arquitectures Avançades. Universitat de Girona online: Consultat durant l'abril de 2007 http://pserv.udg.edu/Portal/DesktopModules/DocTree/ViewFile.aspx?IdArbre=9621&IdDocument=1

[2] Francesc Castro, Mateu Sbert, László Neuman. 2001 Transmittance-based Multipath.

[3] Francesc Castro. Efficient Techniques in Global Line Radiosity. 2002 (Tesis doctoral, Univeristat Politècnica de Catalunya, 2002).

[4] F.Castro, M.Sbert i L.Neumann. 2004. Fast Multipath Radiosity using Hierachical Subscenes. *Computer Graphics Forum, Volume 23, Number 1* pg. 43-53(11)

[5] José González, Antonio González. Evaluation of Alternative Data Speculation Approaches for Superscalar Processors. Universitat Politècnica de Catalunya.

[6] José González, Antonio González. Limits of Instruction Parallelism with Data Speculation. *Proc. of 3rd. Int. Meeting on Vector and Parallel Processing. Porto (Portugal)* pg. 585-598.

[7] Lipasti M. H., Wilkerson C. B. i Shen J.P. 1996. Value Locality and Data Speculation. 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 138-147.

[8] MGF Parser and Examples. Consultat durant el febrer de 2007 http://radsite.lbl.gov/mgf/

[9] Jordi Pagès Marcó. 1999. Simulador configurable d'un processador superescalar accessible via web. (Projecte Final de Carrera d'ETIS, 1999, Universitat de Girona)

[10] Joan Puiggalí, Teo Jové, Salvador Salanova, Josep Lluís Marzo. Execution Speed Up Using Speculation Techniques in Computer Clusters. *Published at Proceedings of of International Mediterranean Modelling Multiconference (IMM'06)* pg. 561-568 ISBN 84-690-0726-2. October 2006.

[11] Joan Puiggalí, Teo Jové, Salvador Salanova, Josep Lluís Marzo. 2006. Limits of TLS execution of sequencial programs on clusters. *Published at Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'06)* pg. 12-19. ISBN: 1-56555-308-X. Calgary, Canada.

[12] Joan Puiggali, Teo Jové, Juan Segovia, Josep Lluís Marzo. 2007 Master/Slave Speculative Parallelization Architecture for Computer Clusters. *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA '2007)* .

[13] PVM- Parallel Virtual Machine, http://www.cdm.ornl.gov/pvm/pvm_home.html

[14] Sazeides Y., Vassiliadis S. i Smith J.E. 1996. The performance potential of data dependence speculation and collapsing. *9th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29)*

[15] Sazeides Y i Smith J.E. 1997. The Predictability of Data Values. *30th IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[16] M. Sbert, X. Pueyo, L. Neumann i W. Purgathofer. 1996. Global multipath monte carlo algorithms for radiosity. *The Visual Computer , V. 12* , pg. 47 - 61

[17] M.Sbert. 1997. The use of global random directions to compute Radiosity Global Monte Carlo techniques. (Tesis doctoral, Universitat Politècnica de Catalunya, 1997.)

[18] J.E. Smith, and A.R. Pleszkun. 1985 Implementation of Precise Interrupts in Pipelined Processors, *in Proceedings of the 12th International Symposium on Computer Architecture (ISCA 85)*, pg. 36-44.

[19] J.E. Smith, and A.R. Pleszkun. 1988. *Implementing Precise Interrupts in Pipelined Processors, IEEE Transactions on Computers, vol. 37, no. 5*, pg. 562-573.

[20] Avinash Sodani, Gurindar S. Sohi. 1998 *Understanding the Differences Between Value Prediction and Instruction Reuse. Microarchitecture. MICRO-31. Proceedings. 31st Annual ACM/IEEE International Symposium on Volume , Issue , 30 Nov-2 Dec 1998* pg. 205 – 215.

[21] Albert Trias. 2005. Instal·lació i experiments en una plataforma multicomputador. (Projecte

Final de Carrera d'ETIS, 2005, Universitat de Girona)
[22] VRML 1.0c Specification. Consultat durant el maig de 2007
http://www.web3d.org/x3d/specifications/vrml/VRML1.0/index.html
[23] Yeh T.Y, Patt Y.N., 1991, Two-level adaptive branch prediction. *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture* pg. 51-61.